Web Services Robustness Testing

Ph.D. Thesis

Samer Hanna,
Department of Computer Science,
Durham University

2008

Dedication

This thesis is dedicated to the soul of my father Odeh and to my mother Azizeh who both always encouraged me to study hard in order to get a PhD, to my brothers and sisters for their continuous support, and specially for my sister Eman and my brother Suhail for helping me to get the PhD scholarship which allowed me to achieve my life time dream.

Abstract

Web services are a new paradigm for building software applications that has many advantages over the previous paradigms; however, Web Services are still not widely used because Service Requesters do not trust services that were built by others.

Testing can assuage this problem because it can be used to assess the quality attributes of Web Services. This thesis proposes a framework and presents a proof of concept tool that can be used to test the robustness and other related attributes of a Web Service. The tool can be easily enhanced to assess other quality attributes.

The framework is based on analyzing Web Services Description Language (WSDL) documents of Web Services to find what faults could affect the robustness quality attributes. After that using these faults to build test case generation rules to assess the robustness quality attribute of Web Services.

This framework will give a better understanding of the faults that may affect the robustness quality attribute of Web Services, how these faults are related to the interface or the contract of a Web Service under test, and what testing techniques can be used to detect such faults.

The approach used in this thesis for building test cases for Web Services was used with many examples in order to demonstrate its effectiveness; these examples have shown that the approach and the proof of concept tool are able to assess the robustness of Web Services implementation and Web Services platforms. Four hundred and two test clients were automatically built by the tool, based on the test cases rules, to assess the robustness of these Web Services examples. These test clients detected eleven robustness failures in the Web Services implementations and nine robustness failures in the Web Services platforms.

Also the approach was able to help in comparing the robustness of two different Web Services platforms, namely Axis and GLUE. After deploying the same Web Services in both of these platforms; Axis showed less robustness and security failures than GLUE.

Declaration

The material contained within this thesis has not previously been submitted for a degree at Durham University or any other university. The research reported within this thesis has been conducted by the author unless indicated otherwise.

This work has been documented in part in the following publications:

1) Samer Hanna and Malcolm Munro. (2007). An approach for Specification Based Test Case Generation For Web Services. 5th ACS/IEEE International Conference on Computer Systems and Applications, AICCSA '2007, May 13-16, Jordan, pp. 16-23
2) Samer Hanna and Malcolm Munro. (2007). An Approach for WSDL-based Automated Robustness Testing of Web Services. 16th International Conference on Information Systems Development (ISD2007), August 29-31, Galway, Ireland.
3) Samer Hanna and Malcolm Munro. (2007). Towards Rule-based Quality of Service Assessment of Web Services. Proceedings of the 7th Service-Oriented Software Research Network (SOSoRNet) Workshop on Dependable, Dynamic Distributed Services and Systems, November 6th, University of Leeds, UK, pp. 25-30.
4) Samer Hanna and Malcolm Munro. (2008). Fault-based Web Services Testing. IEEE Proceedings of the 5th International Conference on Information Technology: New Generation (ITNG 2008), April 7th-9th, Las Vegas, Nevada, USA, pp. 471-476.

Copyright Notice

Acknowledgements

First I want to thank God for changing my life and giving me the strength to write this thesis. Also this thesis would not appear in its present form without the assistant and support of the following individuals and universities:

Professor Malcolm Munro, Durham University, who acted as my first supervisor during the whole research period, Prof. Malcolm helped me to organize my ideas and thoughts and also provided me with continuous advice, guidance and encouragement. Also a great friendship has developed between us in these three years that I hope will last forever.

Dr. Marcio Dias, previous lecturer at Durham University, who acted as my second supervisor in the first year of my research before he left Durham University. Dr. Marcio gave me many good ideas for my research direction.

Professor David Budgen, Durham University, who acted as my second supervisor for the second and third years of my research, Prof. David provided me also with invaluable advice in my research specially in the first year PhD viva.

Dr. William Song, Durham University, for giving me guidance through my first year viva and also for preparing for a group of seminars about Semantic Web and Web Services that contributed to my knowledge in this field.

Philadelphia University, in Jordan, for giving me the scholarship for my PhD at Durham University.

Table of Contents

Chapter 1 – Introduction

Chapter 2 - Web Services

Chapter 5 – An Approach to WSDL-based Robustness Assessment of Web Services

Chapter 6 – WS-Robust: Web Service Robustness Testing Tool

Chapter 7 – Evaluation

Tables

Listings

Chapter 1

Introduction

1.1 Introduction

Web Services are a new paradigm for building distributed software applications. They have many advantages over previous paradigms such as increasing the interoperability between heterogeneous applications and facilitate sharing data and information between an enterprise, its branches and customers, even if they are using a different platform, programming language or operating system.

However, the Web Services paradigm is still not widely adopted by companies and individuals because of the trustworthiness challenge. In the Web Services paradigm, the Service Requester uses a Web Service implementation written by the Service Provider. It is lack of trust in using software written by others that causes the trustworthiness problem between the Service Requester and Provider.

Testing is one aspect of increasing the Service Requesters trust by helping them to automatically assess the robustness quality attribute of a Web Service based on its interface or contract. The Service Requester may be a human, software, or another Web Service. However, in this thesis, the Service Requester is considered only as a human.

This thesis aims to generate test cases to assess the robustness quality attributes of Web Services. The platform, where a Web Service implementation is deployed, may intercept the request message and it is for this reason that each test case specifies if it

aims at detecting a robustness fault in the Web Service implementation or the Web Service platform.

This thesis approach to Web Service testing has proven to be useful by achieving the following results:

- Detecting robustness faults in many Web Services implementations and platforms (see Chapter 7).
- Comparing the robustness of two Web Service Platforms (see Chapter 7).

This chapter will give an introduction about Web Services, Testing, and Web Service testing and also discuss the objectives and the contributions of this thesis.

1.2 Web Services

Web Services (W3C, 2004a) (Ferris & Farrell, 2003) are a new paradigm in building software applications based on the Internet and open standards. This paradigm has changed the way we look at the Internet from being a repository of data into a repository of Services (Zhang & Zhang, 2005c).

By using Web Services, companies can ensure that their applications will communicate with those of their business partners and customers. Web Services now are the basis of many Service Oriented Computing (SOC) (Huhns & Singh, 2005) applications. Spending on Web Service projects has been estimated to reach $11 billion by 2008 (Leavitt, 2004), and in the next 10 years Web Services will become the dominant distributed computing architecture (Zimmermann, 2003).

Web Services are an implementation or realization of the Service Oriented Architecture (SOA) (Huhns & Singh, 2005). While the previous paradigms depend on

components or objects, the means of building software applications in SOA are Services.

A SOA consists of three roles, namely:

- **Service Requester (Service Consumer)**: Is the distributed application builder (a person).
- **Service Provider**: Develops and implements a Web Service.
- **Service Registry**: Stores meta data about Web Services such as the Provider name and the location of the contract.

The Service Provider publishes a contract (description of their Web Service) to the Service Registry. The Service Requester searches the Service Registry for Web Services that accomplish a certain requirement. Once the Service Registry finds the right Web Service it returns the Service information to the Service Requester, which in turn uses this information to bind to the Web Service.

1.2.1 Web Service Advantages and Challenges

Web Services have many advantages such as:

- Increasing the reusability and consequently reducing the time and cost required to build a Web based distributed application.
- Facilitating the communication between heterogeneous applications over the Internet.
- Based on open standards.

However, Web Services face some problems and the following discusses some of these problems:

1) **The trustworthiness problem**: The Service Requester can only see the contract (WSDL) of a Web Service but not the source code. This fact has caused the Web Service trustworthiness problem because Service Requesters do not trust Web Services that were implemented by others without seeing the source code of the Web Service. Tsai (Tsai et al. 2005a) mentioned that this problem is limiting the growth of Web Service applications and that these applications will not grow unless researchers face this trustworthiness challenge.

Zhang (Zhang, 2005a) stated that the current methods and technologies simply cannot ensure Web Service trustworthiness and that for Web Services to grow, researchers must not wait to address this challenge.

2) **The selection problem**: Service Requesters have no criteria to choose between Web Services that accomplish the same task. Zhang (Zhang, 2004a) stated that it is a big challenge to choose the most appropriate Web Service from a "sea of unpredictable Web Services".

The reason for these problems and challenges is that the WSDL contract of a Web Service describes the operation or the function that a Web Service provides and how to bind to this Service. However, it does not describe the non functional quality attributes such as *robustness*, *reliability* or *performance.*

3) **Vulnerability to invalid inputs by malicious Service Requesters**: Since Web Services are advertised in the Internet then any Service Requester can access this Web Service and some of these might be malicious Requesters that aim to harm the Web Service or gain unauthorized access to certain information by providing invalid or malicious input.

Input manipulation vulnerability is 59.16% of the overall Web Services vulnerabilities (YU, et al. 2006) and that is why Web Services should be tested against this kind of fault to assess if a Web service is vulnerable to input manipulation attacks in order to increase Web service trustworthiness.

Myers (Myers, 1979) mentioned that testing that a program does what it is suppose to do is only half the battle, the other half is to test whether the program does what is not supposed to do. In other words, to check if a program is vulnerable to invalid input.

This thesis will use testing to give an approach to solve these Service Requester and Service Provider problems. This thesis applies the traditional input validation testing techniques to Web Services.

1.3 Software Testing

Software Testing (Harrold, 2000) (Jorgensen, 2002) is a Software Engineering technique that is mainly used to detect faults and assess the quality attributes in a software system and to demonstrate that the actual program behavior will conform to the expected behavior. Studies indicate that more than fifty percent of the cost to develop software systems is devoted to testing and that the percentage is significantly higher for critical systems (Osterweil, 1996) (Harrold, 2000).

Testing techniques can be divided into black box and white box depending on the availability of the source code; if test data is generated depending on the source code then a testing technique belongs to *white box* testing, while if the source code is unavailable, and the tester only cares about the behavior of the system under test rather than how it was built, then a testing technique belongs to *black box* testing.

Examples of black box testing techniques are: *boundary value testing* (Jorgensen, 2002), *equivalent partitioning* (Myers, 1979) and *syntax testing* (Beizer, 1990). Example of white box testing are *path testing, data flow testing* and *slice-based testing* (Jorgensen, 2002).

1.3.1 Quality Attributes

A quality attribute (sometimes called property) is defined as the software component characteristic that the developers need to understand in order to integrate the software component with the system under development (Korel, 1999). Examples of a quality attribute are: *dependability, performance, security,* and *testability.*

The quality attribute that this thesis is concerned with is *robustness* which is a sub-attribute of *reliability* (Adrion et al. 1982), which in turn is a sub-attribute of *dependability* (Avizienis et al. 2004) and *trustworthiness* (Zhang, 2005a). Robustness is defined as "the degree to which a software component functions correctly in the presence of invalid inputs or stressful environment conditions" (IEEE, 1990).

1.3.2 Difficulties of Software Testing and Quality Attributes

Software testing and quality attributes have many difficulties and challenges such as:

- Not all the quality attributes are quantifiable.
- There is no agreement between researchers about the relationships between quality attributes, for example, according to (Boehm et al. 1976) the *reliability* quality attribute includes the sub attributes: *self-containedness, accuracy,*

completeness, robustness/integrity and consistency, while according to (Adrion, et al. 1982) it includes *adequacy* and *robustness* sub-attributes.

- There is no agreement about what testing techniques can be used to assess certain quality attributes.
- There is no agreement about what faults may affect certain quality attributes.
- Quality attributes are different for different applications and prospective.

1.4 Web Services Testing

The trustworthiness of Web-Service software is considered the paramount factor that will decide the success of the Web Services paradigm (Zhang & Zhang. 2005c). Software testing is used in this thesis in providing an approach that addresses part of this trustworthiness challenge.

Since testing is performed to support quality assurance then it is normal to use it with Web Services in order to increase their quality and hence increase the Service Requester's and the Service Provider's trust.

The confidence of the Service Requesters of a Web Service will increase or decrease according to the test results. This will help the Service Requesters to choose between Web Services doing the same task.

Using testing to assess the quality attributes of Web Services has many advantages such as:

- Increase the Web Services trustworthiness by the Service Requesters and Providers and hence increase the usage of Web Services to build software applications.

- Help the Service Requesters to choose between Web Services that accomplish the same task depending on the quality attributes that concerns each Service Requester.
- Since Web Services are loosely coupled, when a fault is detected in a certain Web Service in an application then this Web Service can be replaced with another one that accomplishes the same task without affecting the application.
- Help the Service Providers to detect faults in their Web Services before publishing them.
- Help the Service Provider to make sure that his Web Service will survive against attacks by malicious Service Requesters.
- The Service Provider may change the code of his Web Service after publishing it, so regression testing can be used to solve this problem.

However, Web Services testing still face many difficulties such as:

- There is lack of technologies for Web Services verification (Zhang & Zhang, 2005c).
- Current methods and technologies cannot ensure Web Service trustworthiness (Zhang, 2005a) (Tsai, et al. 2005b).
- Due to specific properties of Web Services, the existing traditional software testing techniques deserve modification to make them suitable for the domain of Web Services (Zhang, 2005c)
- New software testing techniques are required to perform effective testing on Web Services(Zhang, 2005c)

- Web Services are based on relatively new open standards such as XML, WSDL and SOAP, while traditional testing techniques were developed earlier than those standards and hence those techniques must be modified to make them work with the new characteristics introduced by the Web Service standards. In other words, the current testing techniques can not merely be applied to Web Services (Zhang & Zhang, 2005b).
- Unavailability of the source code of a Web Service to the Service Requesters i.e. all the test done by the Requester is black box.
- Testing Web Services is very expensive because it consumes significant cast and bandwidth (Zhang & Zhang, 2005b).
- After analyzing WSDL documents for many Web Services it has been found that the descriptions provided for the input parameters can be used to improve test case generation for the Web Service (increase *testability).*

1.5 The Proposed Method of Web Services Robustness Testing

This thesis proposes a method to assess the robustness quality attribute of Web Services. The method focuses on the robustness faults that may lead to robustness failures rather than focusing on whether a Web Service produces the correct response. An exceptional input that is based on the information inside WSDL will be fed to the Web Service under test and the response of this Web Service will be analyzed by a tool to detect robustness failures.

The approach proposed in this thesis for Web Services robustness testing depends on a model that will be described in chapter 5. The proposed model is based on the following general steps:

1. Analyzing WSDL documents to know what faults may affect the robustness quality attribute of Web Services, specifically, the XML Schema specification (W3C, 2004b) (W3C, 2004c) of the input parameters datatype.
2. Analyzing what testing techniques can be used to assess those faults.
3. Analyzing how test data and test cases can be generated, to assess robustness quality attributes, based on step 1 and step 2.

The proposed approach of automated WSDL based robustness testing has many advantages such as:

1. Automating the process of generating test cases to assess the robustness quality attribute of Web Services
2. Addressing the Service Requester's trustworthiness problem discussed in section 1.2.2 by assessing the robustness quality attribute of Web Services.
3. Facilitate discovering faults in Web Services before they result in significant failure.
4. Addressing the Service Requester's selection problem discussed in section 1.2.2 by giving the Requester the robustness criteria to choose between Web Services that accomplish the same task.
5. Addressing Service Provider's vulnerability to invalid inputs problem (discussed in section 1.2.2) that may lead to security breaches in Web Services.

6. Observing how a Web Service will respond if there are problems in its environment such as the problems caused by the input from other Web Services in the same Web Service composition.
7. Standardizing the process of test case generation by all Service Requesters depending on test case generation rules
8. Participating in solving the problem of the lack of technologies for the verification of Web Services discussed in section 1.4.
9. Participating in addressing the Web Services testing problem of the unavailability of the source code to the Service Requester discussed in section 1.4 by designing test cases based only on WSDL.
10. Participating in addressing the Web Services testing challenge of modifying the traditional software testing techniques to make them work with Web Services.
11. Participating in addressing the Web Services testing challenge of extending the WSDL specification to increase the testability of Web Services.
12. Participating in solving the testing problem of specifying the testing techniques that can be used to assess certain quality attributes.
13. Participating in addressing the problem of specifying the faults that may affect certain quality attributes.
14. Participating in addressing the problem of the relationships among the quality attributes.

1.6 Objective

The main problem that this thesis aims to address is the lack of trust of Web Service by Service Requesters and Providers; this problem will be addressed by providing an approach to assess the robustness quality attribute of Web Services using traditional testing techniques.

Researchers in the field of Web Service testing proposed few models for verification and test case generation for Web Services (Tsai, 2005a). However, most of the models focused mainly on applying certain testing techniques for Web Services without clearly analyzing what faults these testing techniques aim to detect or what are the specific quality attributes that will be assessed. Some of the previous research specifies the quality attribute to be assessed, such as assessing the *reliability* quality attribute in (Zhang, 2004a), but did not analyze the sub-attributes of these quality attributes and how they can be assessed.

The objective of this thesis is to introduce a different approach in that it relates faults, quality attributes, and the WSDL components, in rules for Web Service testing which leads to a greater understanding of the faults that may affect the robustness of Web Services. It also defines what test data could be used to assess those faults and also what other quality attributes may be affected by those faults. The influences from the literature on the research on this thesis are shown in Fig. 1.1.

The ultimate goal of the research is to increase the dependability and trustworthiness of a Web Service by assessing Web Services quality attributes. This goal cannot be accomplished in a single piece of work and needs a number of future years of research;

however, the objective of this thesis is so give an approach that can be used as a start point to achieve that goal.

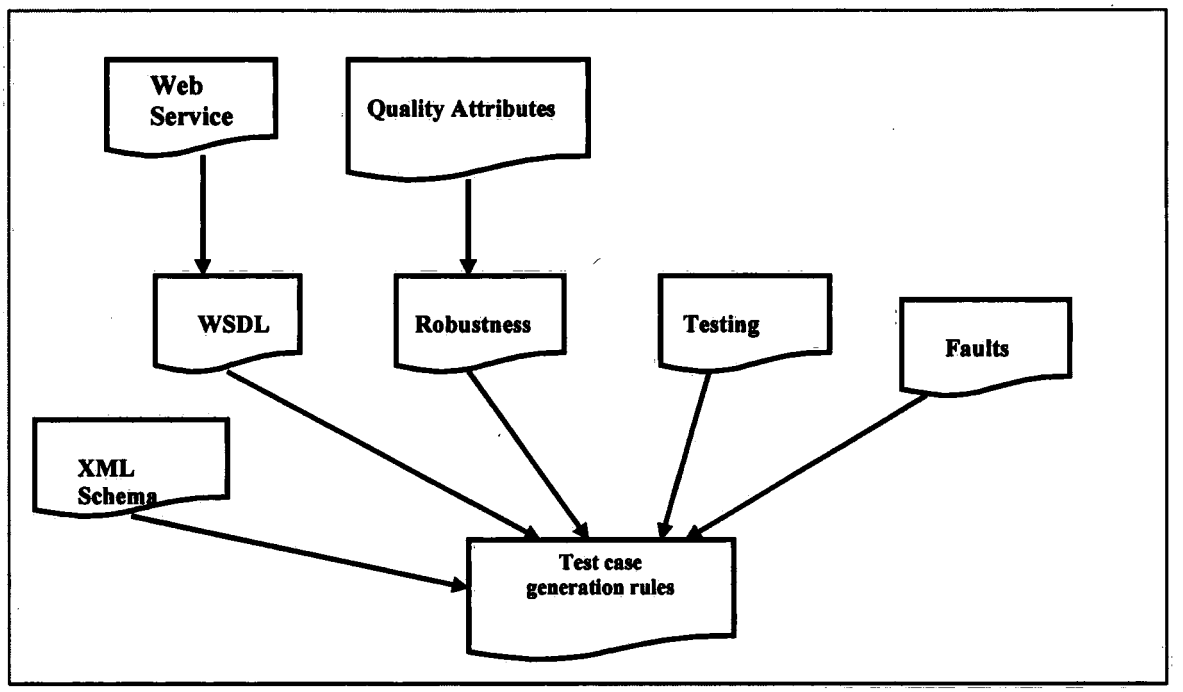


Fig. 1.1. Influence from the Literature

1.7 Contributions

This thesis achieves the following contributions:

1. Developing an approach to assess the robustness quality attributes of a Web Service based only on the specification of the operations' input parameter datatypes inside the WSDL document of the Web Service under test.
2. Detecting robustness and security faults in Web Services implementations and platforms.
3. Analysis of which faults affect the robustness quality attribute of Web Services.

4. Implementing a prototype tool that demonstrates the feasibility of the proposed Web Services robustness testing approach. The tool is able to generate test cases to assess the robustness of Web Service and write a test client depending on these test cases.
5. Analyze the effect of the Web Service platform on the robustness and security quality attributes. A comparison has been made to two platforms by deploying the same Web Services on both of them and then assessing which one of the platforms is more robust and secure using this thesis approach.

1.8 Thesis Structure

This thesis is organized as follows:

Chapter 2 will give a definition to Web Services and the technologies that are used in Web Services. Since test cases are built in this thesis using WSDL and XML Schema, more details will be given for these two W3C specifications for Web Services. Chapter 3 will discuss the traditional testing techniques such as boundary value and robustness testing and also the quality attributes that can be assessed using testing techniques. Chapter 4 will give a comprehensive survey on how other researchers tackled Web Service testing. Chapter 5 will define the proposed method in this thesis that is based on the analysis of the data types in WSDL in order to generate test cases. Chapter 6 will discuss the implementation of the method in Chapter 5. Chapter 7 will evaluate the usefulness of this thesis approach by applying it to many examples or case studies such as the Amazon Web Services. And finally Chapter 8 will give the conclusion of this thesis and also will discuss the future research directions.

1.9 Summary

This chapter gave an introduction to the thesis by:

- Defining Web Services with their advantages and difficulties (section 1.2)
- Defining testing and quality attributes (section 1.3)
- Defining Web Services testing with its advantages and difficulties (section 1.4)
- Describing briefly the model used in this thesis for Web Service testing (section 1.5).
- Describing the objective and the contribution of the thesis (section 1.6 and 1.7)
- Specifying the rest of this thesis structure (section 1.8)

The target of this thesis is to introduce a novel approach for Web Service robustness testing that will help in increasing the trustworthiness of Web Service Requesters and Providers in Web Service applications.

Chapter 2

Web Services

2.1 Introduction

This chapter will give definitions of Web Services, Service Oriented Architecture (SOA), and the open standards that enable a Web Service to implement SOA such as XML, XML Schema, SOAP and WSDL.

2.2 Service Oriented Architecture (SOA)

The software architecture of a computing system is the structure which comprises software components, the external properties of those components, and the relationships among them (Bass et al. 2003). SOA is defined as an approach to building software systems that is based on loosely coupled components (services) that have been described in a uniform way and that can be discovered and composed (Erl, 2006). Another definition is that SOA is a pattern where all software components are modeled as service, where components are functional units that are visible for other entities to invoke or consume over the network (Graham et al. 2005).

The SOA concept is needed to enable Service Oriented Computing (SOC). While previous paradigms of building software applications depend on components or objects, the mean of building software applications in SOA are services.

A SOA includes the following components: Service Requester, Service Provider, Registry, and Contract components as shown in fig. 2.1.

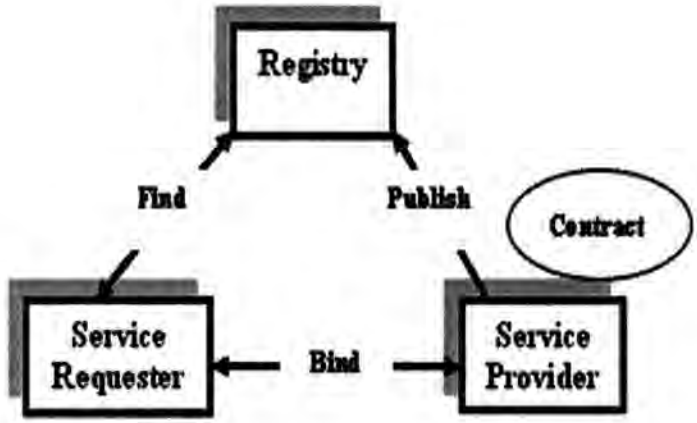


Fig. 2.1. Service Oriented Architecture (SOA)

The relationships between these components are as follows: the Service Provider publishes their Contract or interface in the Registry. Then the Requester of a Service asks the Registry for the Services that matches their criteria. If the Registry has such a Service, it gives the Service Requester information about that Service such as the location of its Contract. Finally the Service Requester can then bind and execute this Web Service using the information in the Contract.

The Contract (interface or description) is important because:

- Service Providers publish information about the location of the Contract inside a Registry.
- Service Requesters use the Contract to bind to the requested Web Service because the Contract describes how a service can be invoked
- The Contract describes all the operations that a Web Service provides.

The Services in a SOA have many characteristics such as (Erl, 2006):

- **Loosely coupled:** the Service Requester should not worry about how a Service was implemented or where the Service is located.

- **Discoverable**: meaning that the Requester of a Service can discover the needed Web Service by asking the Registry as mentioned above.
- **Dynamically bound**: meaning that the Requester of a Service can bind to the Web Service using the information in the Contract at run time.
- **Interoperable:** meaning that a software application can invoke a service even if that Service is on a different platform and written in a different programming language.
- **Network addressable**: meaning the Consumer can invoke a service using a network (usually the Internet).

2.3 Web Service Definition

There is no standard definition of Web Services. The definition has always been under debate. A difficulty with research in this area is the number of definitions of Web Services, many of which are contradictory and imprecise.

Among the many definitions, some of the important ones are:

1. The World Wide Web Consortium (W3C) (W3C. 2004a) (which has managed the evolution of the SOAP and WSDL specifications) defines Web Services as:

   "A software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web Services in a manner prescribed by its description using SOAP, typically conveyed using HTTP with XML serialization in conjunction with other Web-related standards".

2. IBM (IBM 2006) defines Web Services as:

   "A technology that allows applications to communicate with each other in a platform- and programming language- independent manner"

3. Offutt and Xu (Offutt and Xu, 2004) define Web Services as:

   "A Internet-based and modular applications that use Simple Object Access Protocol (SOAP) for communication and data transfer in XML through the Internet".

4. Cerami (Cerami, 2002) defines a Web Service as:

   "Any piece of software that makes it available over the Internet and uses a standardized XML messaging system".

5. Curbera (Curbera, et al. 2002) defines Web Services as:

   "An emerging technology to provide a systematic and extensible framework for application-to-application interaction, built on top of existing Web protocols and based on open XML standards".

6. Zhang (Zhange and Zhang, 2005c) defines Web Services as:

   "Programmable Web applications that are universally accessible using standard Internet protocols"

Clearly, there is no one fixed definition of Web Services, which means that there are different views of the infrastructure that should be considered as a Web Service. However, by observing the above definitions, we notice that there are some characteristics to be considered as a Web Service infrastructure including:

1. Modular - Web Services are usually an aggregation of many loosely coupled and independent Services.

2. Application to application (or machine to machine) interoperable interaction infrastructure, a Web services' main goal is to integrate heterogeneous applications.
3. Use of SOAP, WSDL, and UDDI - Typically Web services use SOAP messages to communicate, and the interface or Contract of a Web Service is described using WSDL, and services descriptions are stored in UDDI.
4. Based on XML - all Web Services technologies are based on XML.
5. The interface is described in a machine processable format.
6. Transport neutral - Usually Web Services transfer over HTTP, but they can transfer over any other transmission protocol.
7. Internet-based - Web Service interactions are done mainly using the Internet but they can be done by any other network

The relationships among these characteristics and the different definitions introduced are summarized in Table 2.1. The table indicates whether we can infer a characteristic (column) based on a particular definition (row).

The symbols shown in the table are:

1. The full circle (●) indicates that the definition explicitly states the characteristic.
2. The symbol (≈) indicates that the definition does not explicitly express that specific characteristic, but the context of the definition suggests it.
3. The empty circle (○) indicates that the characteristic is not included in a specific definition.

Table 2.1. Relations of Web Services Definitions and Characteristics

Characteristic / Definition	Modular	Application Integration	SOAP, WSDL, and UDDI	XML based	Interface is machine-processable	Transport Neutral (typically HTTP)	Internet-based (or any other Network)
W3C (W3C. 2004a)	○	●	●	●	●	●	●
IBM (IBM 2006)	○	●	○	○	○	○	≈
Offutt and Xu (Offutt and Xu, 2004)	●	●	●	●	○	○	●
Cerami (Cerami, 2002)	○	○	○	●	○	○	●
Curbera et al. (Curbera, et al. 2002)	○	●	≈	●	○	≈	≈
Zhang and Zhang (Zhange and Zhang, 2005c)	○	○	≈	○	○	○	●

We can see that W3C (W3C, 2004a) gave the broadest and the most precise definition among the definitions because it specifies all of the characteristics. However, this definition did not specify the modular characteristic that was specified by Offutt and Xu (Offutt and Xu. 2004).

It should also be noted that none of the definitions specified the loosely coupled characteristic of Web Services which is considered one of the main characteristics of SOA.

After analyzing all the definitions, this thesis will use the following definition of Web Services that include all the characteristics mentioned in the above definitions and also the loose coupling characteristic:

> *"Web Services are network (Internet) based modular applications designed to implement SOA, and support interoperable, loosely coupled, integration of heterogeneous applications. Web Services are discovered using UDDI and have an interface (WSDL) that is described in a machine-processable format. Other systems interact with the Web Services in a manner prescribed by its description using SOAP. These SOAP messages (as well as all other technologies of Web Services) are based on XML and typically conveyed using HTTP".*

As an example of using a Web Service; suppose that 3rd person want to build a Web Service based application and part of this application needs to make transactions about products provided by Amazon such as books. Amazon Web Services (Amazon, 2007) is a Web Service interface that is provided by Amazon to enable application builders to invoke the information of Amazon products. A Web application that uses the Amazon Web Service to make transactions on the Amazon books is considered Service Requester and Amazon is the Service Provider.

As another example (Singh & Huhns, 2005) for Web Services and SOC, taken from healthcare domain, suppose that we want to build a Web Service based application for a certain hospital, this application is responsible for purchasing supplies for the hospital. The application should be able to interoperate with the vendor's Services and select the vendor with the best quality of Service criteria such as reliability, performance, and availability.

2.4 Web Services Architecture

Web Services provide platform-independent communication of software Services (resources) across the Internet. While many believe that Web Services are SOA, they are in fact, implementations of SOA. SOA is an architectural concept, an approach to building systems, Web Services, on the other hand, are an implementation of SOA that is based on a set of XML-based technologies such as SOAP and WSDL.

To implement a SOA, Web Services depend on a group of eXtensible Markup Language (XML) (W3C, 2006) standards such as:

Simple Object Access Protocol (SOAP) (W3C, 2007) which plays the role of the messaging protocol for exchanging data between the Service Provider and the Service Requester (application builder). SOAP protocol is considered the core of XML-based distributed computing.

Web Service Description Language (WSDL) (W3C, 2001) which plays the role of the contract that describes the operations provided by a Web Service and how to bind to it.

Universal Description, Discovery, and Integration (UDDI) (OASIS, 2004) which plays the role of a Registry of Web Services descriptions or contracts.

These standards enable Service Requesters to search for Web Services contracts to find a Service that fulfils their requirements, and then use the information inside the contract to communicate with remote Service Providers by using a non-proprietary protocol such as SOAP over Hyper Text Transfer Protocol (HTTP) (Gourley et al. 2002) or other transport protocols.

Since SOAP is the core protocol for distributed computing and it is used in almost all Web Services, and since HTTP is the ubiquitous communication protocol on the Internet that is also used by most Web Services, this thesis will use only SOAP/HTTP for a messaging/transport protocol.

Web Services can be thought of as a layered set of technologies or standards as shown in Fig. 2.2. However, Fig. 2.2 includes only the technologies that are relevant to this thesis. There are other technologies for Web Services such as those that enable Web Service composition; however, those technologies are outside the scope of this thesis.

The layers of the Web Service technologies stack shown in Figure 2.2, are:

1. **Transport Layer:** The base layer of the stack is the transport layer. Since Web Services are basically a messaging mechanism between applications over the Internet, they rely on transport technologies such as HTTP which are used as transportation protocol in the Internet.

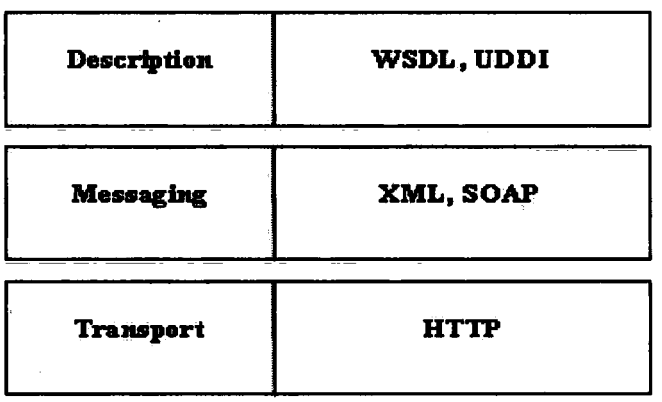


Fig. 2.2. Web Services Technologies Stack

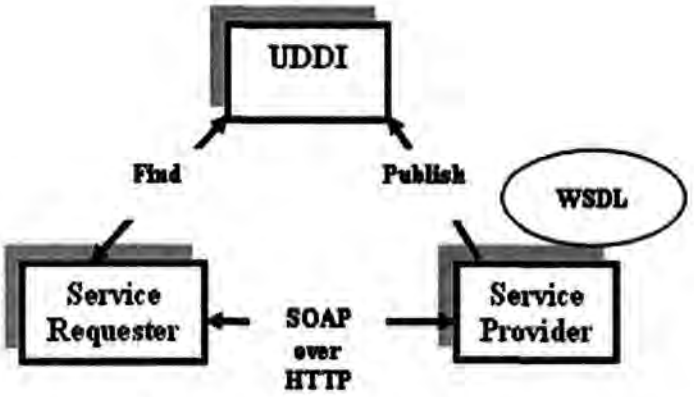


Fig. 2.3. Web Services Architecture

2. **Messaging Layer:** At the messaging layer there are the fundamental Web Services technologies, namely, XML and SOAP. XML will be discussed in section 2.6.1 and SOAP will be discussed in section 2.6.3.
3. **Description Layer:** This layer is responsible for describing a Web Service such as what operations a Web Service provides and how to find it. The technologies at this layer include WSDL and UDDI. WSDL will be discussed in section 2.6.3 and UDDI will be discussed in section 2.6.4.

Applying these Web Service technologies to Fig. 2.1 gives Fig. 2.3.

The Web Service technologies in Fig. 2.2 and Fig. 2.3 enable Web Services to implement SOA as the following:

- WSDL plays the role of the Contract in SOA, UDDI plays the role of the Service Registry, and SOAP plays the messaging protocol that is responsible for binding the Service Requester and the Service Provider.
- Service Providers publish information about their Web Services implementation including WSDL address in the UDDI registry. When a Service Requester needs

to invoke a certain Service provided by a Service Provider, they must do the following:

1. The Service Requester search the UDDI for the Web Services that meet their requirement specification
2. The UDDI registry will tell the Service Requesters the location of the required Web Service and the location of its WSDL
3. The Service Requester will use the information inside WSDL to send a SOAP message as a request to the required Service provided by the Web Service implementation or Service Provider.
4. The Service Provider replies by sending a SOAP response to the Service Requester, if the SOAP request has some errors then the Service Provider replies by a SOAP fault rather than a SOAP response.

2.5 Web Service Invocation

A Web Service can not be accessed directly by Service Requesters but they are accessed by software applications that are written by these Service Requesters (Service Requesters are assumed to be human and not software in this thesis). These software applications are called Web Services-based application (or Client applications) this is the new paradigm of building software application that relies on Services available on the Internet.

Fig. 2.4 shows a model that describes the components that participate in a typical Web Service invocation in a Web Service-based application. Most of the components in this model have already been defined except: client stub, server stub (skeleton) and Web

service container. These terms will be defined first and then a description of the components interaction in the model will be discussed.

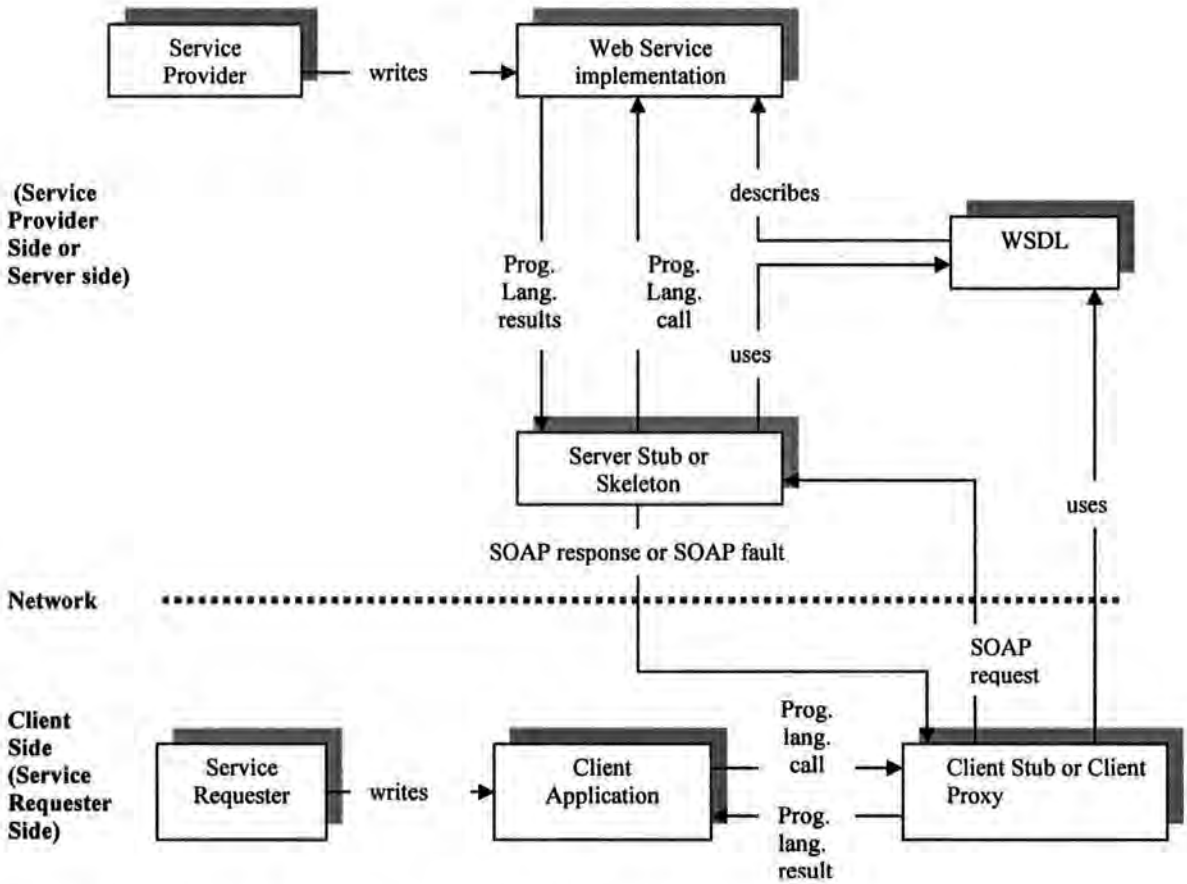


Fig. 2.4. A Model for a Web Service Application

Client stub (sometimes called client proxy), is code (such as Java code) that is generated from WSDL and is responsible for:

- Giving Service Requesters an API that mirrors the Web Service operations inside WSDL.
- Taking a request (or call) in application specific data (such as in a Java datatype) from the client application and converting this request into SOAP

request. This process of converting application data to XML (SOAP) is called marshalling.

- Receiving SOAP responses and converting them to an application specific data that is understood by the client application

Server stub (or skeleton), is a code that is responsible for:

- Receiving a SOAP request from client stub (using HTTP) and converting it into a form that is suitable for the Web Service implementation. This process of converting XML (SOAP) to application specific data is called unmarshalling.
- Converting the response from the Web Service implementation into a SOAP response message (or fault message in the case where the Web Service implementation raised an exception or anything went wrong)

Web Service container (or server) provides a hosting environment for Web Service source code and the middleware (or SOAP implementation) such as Axis (Apache Software Foundation, 2007). The container or the server is the first to receive the HTTP SOAP request from the Service Requester, the server then decides what to do with this request message according to a field inside HTTP POST called SOAPAction.

An example of a Web service container is apache tomcat (Apache Software Foundation, 2006).

The process of invoking a certain operation (see Fig. 2.4), provided by a Web Service implementation, includes the following steps:

1. The client application calls the client stub using an application specific datatype (depending on the programming language this application is written in)

2. The client stub will convert this local invocation into a SOAP request (marshalling)
3. The SOAP request is sent over the Internet (using HTTP) to the required Web Service container
4. The Web Service container (server) receives this SOAP request and then hands it to the skeleton (server stub).
5. The skeleton converts the SOAP request into an application specific data and sends it to the Web Service implementation (depending on the programming language the Web Service implementation is written by).
6. The Web Service implementation performs the requested operation that it was asked to perform by the skeleton.
7. The result of this operation will be handed to the skeleton.
8. The skeleton converts this application specific result into a SOAP response (or SOAP fault if the Web Service implementation raised an exception or anything else went wrong)
9. The SOAP response (or SOAP fault) message is sent to the client stub using the Internet (over HTTP).
10. The client stub converts the information inside the SOAP response (or SOAP fault) message into an application specific information (that can be understood by the client application) and sends it to the client application.

There are many tools that can create a client stub and a server stub based on WSDL, and manages the creating and sending of SOAP messages over the Internet. These tools are called SOAP-based Web Service platforms or SOAP engines. An example of these

tools is Apache Axis (Apache Software Foundation, 2007) and GLUE (WebMethods, 2007).

2.6 Web Services Standards

This section will present more details for the Web Service standards that are related to this thesis, namely, XML, XML Schema, WSDL, SOAP, and UDDI.

2.6.1 XML

XML (Extensible Markup Language) has the following characteristics:

- Is based on human readable tags
- Extensible language: because 3rd person can define any number of tags he wants.
- Cross-platform.
- Hierarchical: because each element of the XML element can have any number of child elements under it.

List 2.1 is an example of a XML document that describes books, this example will be used to clarify the usage of XML;

XML used for:

- **Structuring and describing data**: in List 2.1 we notice how the information about books are structured and described in a hierarchical way; each *books* element contains the *book* sub-element and many sub-elements such as *ISBN* and title.
- **Storing data**: List 2.1 is considered a way of storing data about the details of books.

- **Exchanging data:** XML is used to exchange data between otherwise incompatible applications or software systems, in other words, XML is a way of connecting heterogeneous applications. In List 2.1 example, when any application receives the *books* information, that application will understand or interpret this information no matter what programming language or platform is used in the receiving application.

Other characteristic of a XML document is that it is possible to use *namespace* for the naming of element and attributes. This is because XML may be used for data exchange and different applications that exchange XML document may use the same name for an element or attribute. XML *namespaces* was introduced to solve this problem by distinguishing between those elements and attributes and also grouping each set of elements and attributes so that they can easily be reused in other documents. An example of an XML document with a namespace is given in List 2.2.

List 2.2, the *year* element is now qualified with the namespace *ns1* to distinguish it from probable other *year* elements in different documents.

If an element is unqualified with a namespace then it uses the *default* namespace which is the namespace that does not have any prefix; in List 2.2 the default namespace is http://www.dur.ac.uk and it is used to qualify all other elements and their sub elements except the *year* element because it has a unique namespace.

2.6.2 XML Schema

XML Schema and Document Type Definition (DTD) (Harold and Means, 2004) are two ways to specify the legal or acceptable building blocks (elements and attributes) of an XML document.

The DTD has limited support for data types and solving this and other problems has led to the introduction of XML Schema by W3C (W3C, 2004b). XML Schema became a W3C Recommendation in 2001 and is used to:

```
<?xml version="1.0" encoding="UTF-8"?>
<books>
    <book topic="Java Programming Language">
        <isbn>0-13-129014-2</isbn>
        <title>Java How To Program Sixth Edition</title>
        <author name="H. M. Deitel" type="fisrt_author"/>
        <author name="P. J. Deitel" type="second_author"/>
        <year>2005</year>
        <notes>used as a tutorial for Java language</notes>
        <publisher>Pearson Education International
        </publisher>
        <location>United States</location>
        <owner>
            <name>S. Hanna</name>
            <email>samer.hanna@dur.ac.uk</email>
        </owner>
    </book>
    <book topic="SOA">
        <isbn>0-13-185858-0</isbn>
        <title>Service-Oriented Architecture Concepts,
      Technology
        </title>
        <author name="T. Erl" type="firsrt_author"/>
        <year>2005</year>
        <notes>Good book to understand SOA concepts</notes>
        <publisher>PRENTICE HALL</publisher>
        <location>United States</location>
        <owner>
            <name>S. Hanna</name>
            <email>samer.hanna@dur.ac.uk</email>
        </owner>
    </book>
```

List 2.1. XML Document Example

- Put constraints on the elements and attributes that can be in an XML document instance.
- Define the relations (structure) between the elements.
- Define the datatypes associated with the *elements* and *attributes*.

```
<?xml version="1.0" encoding="UTF-8"?>
<books xmlns:ns1="http://www.dur.ac.uk/Year"
 xmlns = "http://www.dur.ac.uk">
    <book topic="Java Programming Language">
        <isbn>0-13-129014-2</isbn>
        <title>Java How To Program Sixth Edition</title>
        <author name="H. M. Deitel" type="fisrt_author"/>
        <author name="P. J. Deitel" type="second_author"/>
        <ns1:year>2005</ns1:year>
        <notes>used as a tutorial for Java
      language</notes>
        <publisher>Pearson Education
      International</publisher>
        <location>United States</location>
        <owner>
            <name>S. Hanna</name>
            <email>samer.hanna@dur.ac.uk</email>
        </owner>
    </book>
    <book topic="SOA">
        <isbn>0-13-185858-0</isbn>
        <title>Service-Oriented Architecture Concepts,
      Technology</title>
        <author name="T. Erl" type="firsrt_author"/>
        <ns1:year>2005</ns1:year>
        <notes>Good book to understand SOA
      concepts</notes>
        <publisher>PRENTICE HALL</publisher>
        <location>United States</location>
        <owner>
```

List 2.2. An XML Document with *namespace*

List 2.3 is XML Schema for the XML document in List 2.1 contains examples of many XML schema components such as *datatypes*, *constraining facets*, and *restricting elements*. List 2.3 will be used through the discussion of the XML Schema components.

According to W3C (W3C, 2004c) XML schema datatypes can be categorized into *simple* datatypes and *complex* datatypes:

2.6.2.1 Simple Datatypes:

Simple datatypes include:

Built-in primitive datatypes: an example of a built-in primitive simple datatype in List 2.3 is xsd:*string* (xsd stands for XML Schema Datatype). Fig 2.5 (W3C, 2004c) gives more examples of these datatypes such as *float*, *time* and *anyURI*.

Derived from built-in primitive datatypes: these datatypes are derived from the built-in primitive datatypes by applying some default constraints, for example nonPositiveInteger (Fig 2.5) is derived from integer by restricting the value space of integer to only negative numbers. Fig. 2.5 gives a hierarchy of XML Schema' built-in and derived from built-in datatypes.

User-derived datatypes: User-derived datatypes are simple datatypes derived by restricting a *base* datatype (which can be a built-in primitive or derived from primitive datatypes) using *constraining facets* (See Table 2.2). As an example of a user-derived datatype in List 2.3 is the *Publisher* datatype which has restricted the values that are a *string* base datatype using the *enumeration* constraining facet.

List datatypes: consists of a finite length sequence of values of built-in, derived from built-in or user derived datatypes. All the values of a list need to have the same datatype.

Union datatypes: the union of the values of one or more datatypes.

2.6.2.2 Complex datatypes

Complex datatypes consist of one or more elements and attributes of simple datatypes. Examples of complex datatypes in List 2.3 are: Books, book, Owner and Author. For example, *Books* datatype is a *sequence* of *book* complex datatype. Where *sequence*, *choice* and *all* (W3C, 2004b) are used to put restrictions on the element inside a complex datatype as described in Table 2.3.

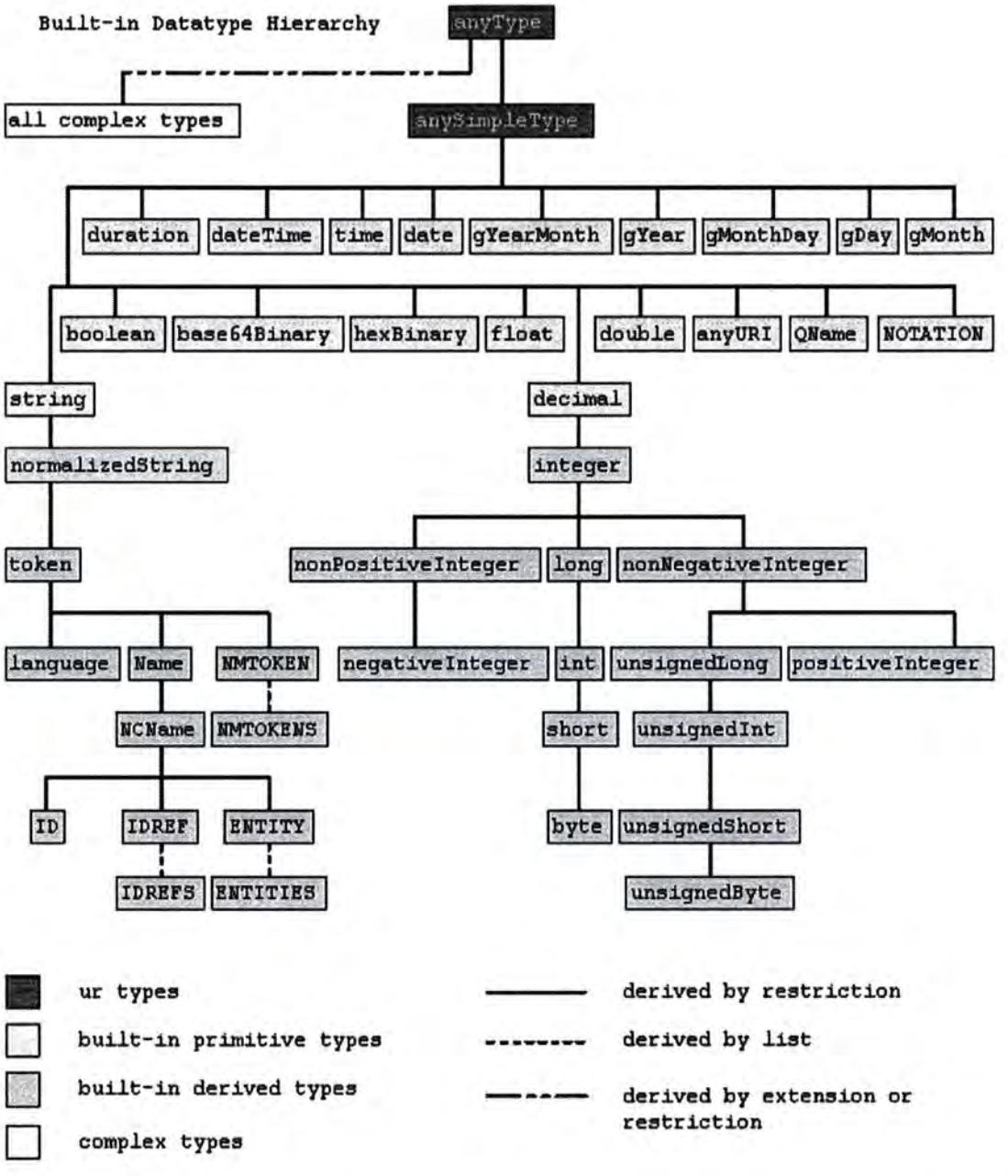


Fig. 2.5. Hierarchy of XML Schema Built-in and Derived from Built-in Datatypes

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
            targetNamespace="http://www.dur.ac.uk/samer.hanna"
            xmlns:bookns="http://www.dur.ac.uk/samer.hanna"
  <xsd:annotation>
    <xsd:documentation xml:lang="en">
      Schema for books.
    </xsd:documentation>
  </xsd:annotation>
  <xsd:element name="books" type="bookns:Books"/>
  <xsd:complexType name="Books">
    <xsd:sequence>
       <xsd:element name="book" minOccurs="0"
      maxOccurs="unbounded">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="isbn" type="xsd:string"
                  minOccurs="1" maxOccurs="1"/>
            <xsd:element name="title" type="xsd:string"/>
            <xsd:element name="author" type="bookns:Author"
                  minOccurs="1" maxOccurs="10"/>
            <xsd:element name="year" type="xsd:positiveInteger"/>
            <xsd:element name="notes" type="xsd:string"/>
            <xsd:element name="imagePath" type="xsd:string"/>
            <xsd:element name="publisher"
            type="bookns:Publisher"/>
            <xsd:element name="location" type="xsd:string"/>
            <xsd:element name="owner" type="bookns:Owner"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="Owner">
    <xsd:sequence>
      <xsd:element name="name" type="xsd:string"/>
      <xsd:element name="email" type="xsd:string"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="Author">
    <xsd:attribute name="name" type="xsd:string"/>
    <xsd:attribute name="type"    type="xsd:string"/>
  </xsd:complexType>
  <xsd:simpleType name="Publisher">
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="Pearson Education Internationa"/>
      <xsd:enumeration value="PRENTICE HALL"/>
      <xsd:enumeration value="John Wiley &amp;amp; Sons"/>
      <xsd:enumeration value="Sams Publishing"/>
      <xsd:enumeration value="Wrox"/>
    </xsd:restriction>
  </xsd:simpleType>
```

List 2.3. XML Schema for the XML Document in List 2.1.

Table 2.2. Definitions of Constraining Facets

Constraining Facet	Definition
Length, minLength, maxLength	Specifies the exact, minimum, and maximum number of units of length, where units of length are: - *character* in case of string or derived from string datatypes - *octets* (bytes) in case of *hexBinary* and *base64Binary*
minInclusive, minExclusive, maxInclusive, maxExclusive	Specifies the inclusive lower bound, exclusive lower bound, inclusive upper bound, and exclusive upper bound for *ordered* datatypes (W3C, 2004c).
enumeration	Constrains the possible values to a specified set or list of values
pattern	A regular expression that specifies the syntax of the allowed value
totalDigits	Constrains the maximum number of decimal digits in a *decimal* datatype
fractionDigits	Constrains the maximum number of decimal digits in the fractional part of a *decimal* datatype
whiteSpace	Defines the way the white spaces are handled in string or derived from string datatypes.

Table 2.3. XML Schema Components Used to Restrict the Order and Occurrence of Elements in a Complex Datatype

XML Schema component (element)	Description
sequence	The child elements must appear strictly in the same order and each child element can be absent or occurs any number of times.
choice	Only one of the child elements is allowed to appear
all	The child elements are allowed to appear in any order and each element can be either absent or occur just one time.

2.6.3 Web Services Description Language (WSDL)

WSDL is a formal, human readable, XML-based interface or specification for describing the capabilities of a Web Service including:

1) What a Web Service can do: This include:

- All the operations or methods that are provided by a Web Service.
- The input and output messages for those operations.
- The parameters that these input and output messages assume.

2) How it can be invoked.

3) Where the Web Service resides

4) What datatypes a Web Service uses

WSDL uses XML elements and attributes to describe these features of a Web Service. Fig. 2.6 is a semantic data model that describes these elements and attributes, and also how they are related.

Table 2.4 is a data dictionary (Sommerville, 2004) for WSDL entities (elements and attributes) and their relations that are described in the model of Fig. 2.6. These have the following conventions:

- The dash between two entities in the table (e.g. *service-port)* is used to declare that there is a relation between these two entities.
- When the same name is given to different attributes in the model (such as the *name* attribute) then the data dictionary use the element that this attribute belongs to in order to know which attribute is meant (e.g. *service name, binding type*).
- The model in Fig. 2.6 is close to the Entity-relationship and UML models but used for XML elements and attributes.

The root element of any WSDL document is the *definitions* element. It consists of five main elements, namely: *types, message, portType, binding, and service*. These main elements reference each other using special *attributes* inside each of them as follows:

- *service* element reference *binding* element using the *port*'s binding attribute, where *port* is sub-element of *service*.
- *binding* element reference *portType* element using the *binding type* attributes.
- *portType* element reference the *messages* element using the *name* attribute of the *operations*' *input*, *output*, and *fault* sub-element.
- *message* element reference *types* element using *part* attribute.

***portType* Element:**

The *portType* is considered the main element inside WSDL because it can be used to describe to the Service Requester the operations provided by the Web Service and what the *input* and the *output message* each *operation* expects. The *portType* of List 2.4 is a description of the operations that are provided by the *Triangle* Web Service. The first operation called *triangleType*. When the Service Requester analyzes this *portType*, they can conclude the following information about this operation:

- This *operation* has three *input* parameters, the ordered names of these parametes are *a, b,* and *c*.
- The input message to this operation is *impl:triangleTypeRequest* where *impl* is a namespace that is declared elsewhere inside WSDL.
- The *output* message for this operation called *impl:triangleTypeResponse*.
- There is no *fault* message for this operation.

The same information about the other operation triangleArea can also be obtained using *portType*.

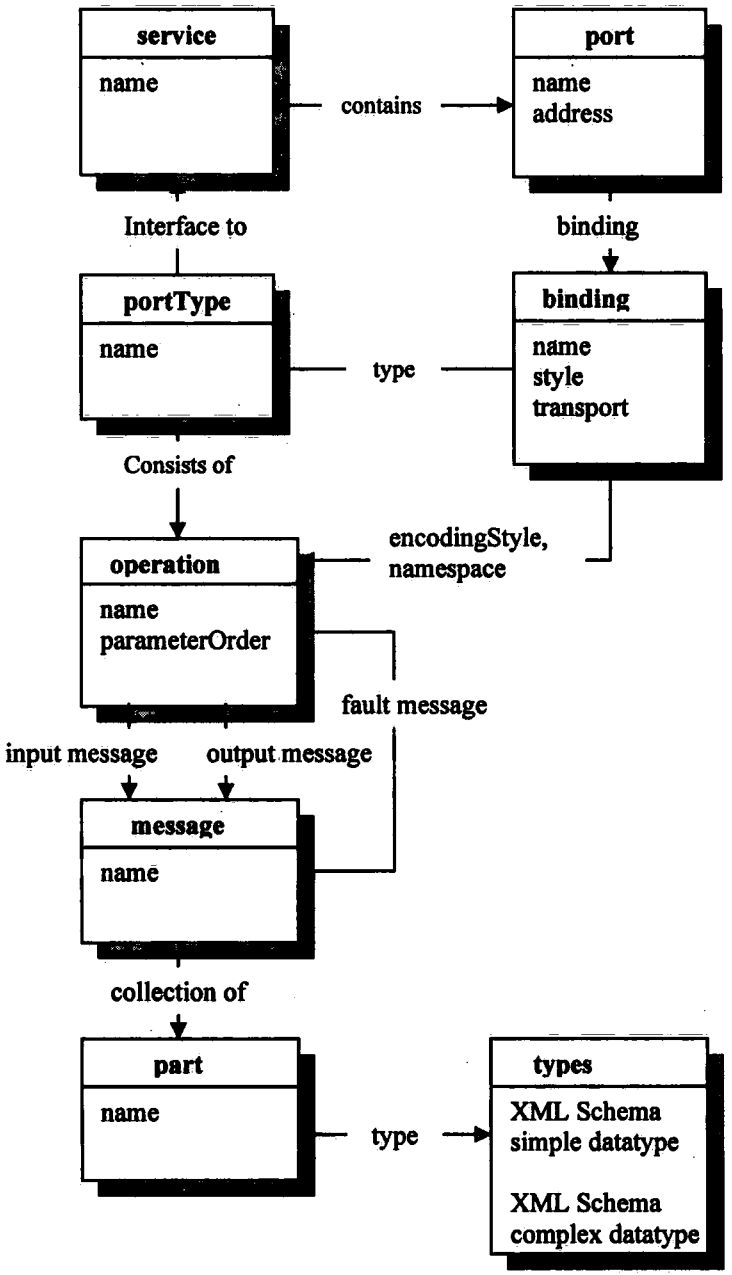


Fig. 2.6. Semantic Data Model for WSDL

Table 2.4 (a). Data dictionary for WSDL Elements, Attributes and their Relations

Name	Description	Type
address	Protocol specific data for the actual location of a Web Service	Attribute
binding	Describes to the Service Requester how to invoke *operations*	Element
binding	The attribute that is used by *port* to reference or associate to a particular *binding* element. Each *port* associates a protocol-specific address to an individual *binding*	Attribute and relation
binding name	A unique name for a specific *binding*	Attribute
binding type	Each *binding* describes a *portType* and the *binding*'s *type* attribute is used to specify which *portType* this *binding* describes.	Attribute and Relation
encodingStyle	A URI (http://www.w3c.org/2003/soap-encoding in SOAP 1.2) that define the rules of encoding the data inside the SOAP messages that are used to invoke a certain *operation.* These rules are used for the purpose of data marshalling during a RPC.	Attribute and Relation
fault message	Specifies the *fault message* of a specific operation, each operation may have 0 or more *fault* messages	Attribute and Relation
input message	Specifies the *input message* name to a specific *operation*, each *operation* may have 0 or 1 *input message*	Attribute and Relation
message	Describes the data travel from one endpoint to another.	Element
message name	The *name* of a specific *request, response* or *fault message*	attribute
message-part	Each *message* is a collection of *parts*	Relation
namespace	A namespace associated with a particular *operation*	Attribute and Relation
output message	The *output message name* of a specific *operation,* each *operation* may have 0 or 1 *output message*	Attribute and Relation
operation	Defines a method on a Web Service	Element
operation name	The name of a specific *operation* or method	Attribute
operation-message	Each *operation* can have three types of messages: *input* message, *output* message and *fault* message.	Relation
parameterOrder	The order of the *parts* that must be used when invoking a *message.*	Attribute
part	Individual parameter for a *message*	Element
part name	The *name* of a *message* parameter	Attribute
part type	The datatype of a specific *part, type* reference a datatype inside *types* element	Attribute and Relation
port	Specifies the address of the endpoint that hosts the Web Service.	Element
port name	A unique name for a *service port*	attribute
portType	Description of the interface of a Web Service that specifies what it can do or what are the *operations* provided by this Web Service	Attribute
portType-binding	Each *portType* has one or more *binding* elements associated with it.	Relation
portType name	A unique *name* of a specific *portType*	Attribute
portType-operation	*A portType* contains a collection of 0 or more *operations*	Relation
portType-service	*portType* is considered an interface to a specific *service*, each service may have 0 or more interfaces *or portTypes*	Relation
service	Specifies where to find the Web Service, *port*, and *binding*	Element
service-port	Each service contains a set of (one or many) *ports* (endpoints)	Relation

Table 2.4 (b)

Name	Description	Type
service name	Each *service* inside WSDL must have a unique name	Attribute
style	Style of invocation the *binding* use which is either Remote Procedure Call (*rpc*) or XML document (*document*) (in this thesis only *rpc* is considered)	Attribute
transport	Specifies what is the transport protocol (such as HTTP or SMTP)	Attribute
type name	The *name* of a specific parameter of a *message*	Attribute
types	Defines the datatypes used in WSDL, the defaults in XML Schema datatypes and it should be the only datatypes used in order to build an interoperable Web Services.	Element
Xml Schema complex datatype	(see section 2.6.2.2)	Element
XML Schema simple datatype	(see section 2.6.2.1)	Element

```
<wsdl:portType name="Triangle">
   <wsdl:operation name="triangleType" parameterOrder="a b c">
    <wsdl:input message="impl:triangleTypeRequest" name="triangleTypeRequest"/>
    <wsdl:output message="impl:triangleTypeResponse"
name="triangleTypeResponse"/>
   </wsdl:operation>
   <wsdl:operation name="triangleArea" parameterOrder="a b c">
    <wsdl:input message="impl:triangleAreaRequest" name="triangleAreaRequest"/>
    <wsdl:output message="impl:triangleAreaResponse"
name="triangleAreaResponse"/>
   </wsdl:operation>
 </wsdl:portType>
```

List 2.4. An Example of a WSDL *portType* Element

The operations of List 2.4 has an input and output message but no fault message, this kind of operation mode or message exchange pattern is called a Request-Response. There are four types of operation modes depending on the combinations of input, output and fault message (Graham, 2005), namely:

Request-Response operations: this is the most common style or mode of operation found on WSDL document. This style of operation defines an input message (the request), an output message (the response), and an optional fault message.

One-way operations: this mode of operation has only an input message and does not have an output or a fault messages.

Notification operations: this mode of operation has only an output message but not input or fault messages. This mode is similar to One-way but the direction of messages is from the Service Provider to the Service Requester to notify them of some event.

Solicit-Response operations: this mode is similar to Notification operation but the Service Requester sends an input message (which is considered as a response) when they receive the notification or an output message from the Service Provider. This style has input, output and optional fault messages similar to the request-response style, however, the response message is the first sub-element of the operation and is then followed by the input message and the optional fault message.

The operation mode in Fig. 2.4 is Request-Response operation mode and this mode will be the only one of the operations modes that will be used in this thesis for two reasons:

1. It is the most common style of operations found in WSDL.
2. A response message from the Service Provider is needed to assess the robustness of a certain Web Service using the approach in this thesis.

***binding* Element:**

The other important element inside WSDL is the *binding* element, the *portType* gives only an abstract description of the operations and the messages while binding describes how these operation transmitted over the network, e.g. using SOAP over HTTP or

SOAP over SMTP. Binding also specifies if the message invocation is RPC or document-centric. List 2.5 shows an example of a binding element.

The following information can be extracted from the binding element in List 2.5:

- The *binding* name is *TriangleSoapBinding*
- The *portType* that this binding associated with is *imple:Triangle* (see List 2.4)
- The style of this binding is *rpc* or remote procedure call as declared by the style attribute (style="rpc").
- The messaging/transport protocol is SOAP over HTTP as declared by the *transport* attribute (*transport=http://schemas.xmlsoap.org/soap/http*).
- How data are encoded in the SOAP message *body* (see section 2.4.4) for the SOAP message used in this binding (see the *wsdlsoap:body* element)
- The operations provided by the Web Service described and the input and output messages of each operation. For example, the *triangleType* operation has *triangleTypeRequest* message as its input message and triangleTypeResponse as its output message.
- The encoding style of the SOAP messages to each operation

***service* Element:**

The service element is a group of *port*s (endpoints), and WSDL may contain more than one service element but conventionally each WSDL document contains a single *service* element. List 2.6 is an example of a *service* element from the same WSDL document of List 2.4 and List 2.5.

The information that can be concluded from the *service* element in List 2.6 includes:

- The *service* name is *TriangleService*
- The *port name* is *Triangle*
- The *binding* that this port associates address to is called *impl:TriangleSoapBinding* (List 2.5)
- The Web Service's location is *http://localhost:8080/axis/Triangle.jws*, so now the binding is associated with a protocol specific (HTTP) data of the location of the Web Service being described.

```
<wsdl:binding name="TriangleSoapBinding" type="impl:Triangle">
  <wsdlsoap:binding style="rpc"
     transport="http://schemas.xmlsoap.org/soap/http"/>
  <wsdl:operation name="triangleType">
   <wsdlsoap:operation soapAction=""/>
   <wsdl:input name="triangleTypeRequest">
    <wsdlsoap:body
     encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
     namespace="http://localhost:8080/axis/Triangle.jws" use="encoded"/>
   </wsdl:input>
   <wsdl:output name="triangleTypeResponse">
    <wsdlsoap:body
     encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
     namespace="http://localhost:8080/axis/Triangle.jws" use="encoded"/>
   </wsdl:output>
  </wsdl:operation>
  <wsdl:operation name="triangleArea">
   <wsdlsoap:operation soapAction=""/>
   <wsdl:input name="triangleAreaRequest">
    <wsdlsoap:body
     encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
     namespace="http://localhost:8080/axis/Triangle.jws" use="encoded"/>
   </wsdl:input>
   <wsdl:output name="triangleAreaResponse">
    <wsdlsoap:body
     encodingStyle=http://schemas.xmlsoap.org/soap/encoding/
     namespace="http://localhost:8080/axis/Triangle.jws" use="encoded"/>
   </wsdl:output>
  </wsdl:operation>
 </wsdl:binding>
```

List 2.5. An Example of a WSDL *binding* Element

```
<wsdl:service name="TriangleService">
  <wsdl:port binding="impl:TriangleSoapBinding" name="Triangle">
   <wsdlsoap:address location="http://localhost:8080/axis/Triangle.jws"/>
  </wsdl:port>
 </wsdl:service>
```

List 2.6. An Example of a WSDL *service* Element

***types* element:**

This element is of special importance to research in this thesis because the approach of test data generation to assess the robustness quality attributes of Web Services that will be discussed in chapter 5 is based on analyzing the datatypes of the parameters in the input messages, and those datatypes are described inside the *types* element of WSDL. An example of a *types* element also from the same WSDL of List 2.4, List 2.5, and List 2.6 is given in List 2.7.

List 2.7 describes two XML Schema simple datatypes (see section 2.4.2.1) that are used somewhere else in the WSDL document to specify that datatype of the parameters to the *input*, *output*, or *fault* messages.

***message* Element:**

A *message* element is used to describe the *input*, *output*, and *fault* messages that travel between the Service Provider and the Service Requester. The *message* element specifies what parameters (*parts*) each message accepts together with that datatypes of these parameters. An example of a message element is given in List 2.8.

List 2.8 from the same WSDL of List 2.4 to List 2.7, describes a message called *triangleAreaRequest* that has three parameters (*parts*) all of them of the simple XML Schema datatype *integerLessThanOrEqualHundred* that was described in List 2.7.

***definitions* Element:**

The *definitions* element is the root element of any WSDL and all other element discussed are sub-element of it. Its element indicates that WSDL is only a group of definitions. *definitions* element also defines the *namespaces* that are used in a WSDL

```
<types>
 <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
       targetNamespace= "http://localhost:8080/axis/Triangle.jws">
  <xsd:simpleType name="integerLessThanOrEqualHundred">
    <xsd:restriction base="xsd:integer">
       <xsd:maxInclusive value="100"/>
        <xsd:minInclusive value = "1"/>
  </xsd:restriction>
  </xsd:simpleType>
  <xsd:simpleType name= "TriangleType-DataType">
      <xsd:restriction base="xsd:string">
       <xsd:enumeration value = "Equilateral"/>
       <xsd:enumeration value = "Scalene"/>
       <xsd:enumeration value = "Isosceles"/>
       <xsd:enumeration value = "Not a triangle"/>
       <xsd:length value = "14"/>
      </xsd:restriction>
  </xsd:simpleType>
 </xsd:schema>
</types>
```

List 2.7. An Example of a WSDL *types* Element

```
<wsdl:message name="triangleAreaRequest">
  <wsdl:part name="a" type=" integerLessThanOrEqualHundred"/>
  <wsdl:part name="b" type=" integerLessThanOrEqualHundred"/>
  <wsdl:part name="c" type=" integerLessThanOrEqualHundred"/>
</wsdl:message>
```

List 2.8. An Example of a WSDL *message* Element

document. A *definitions* element from the same WSDL of List 2.4 to 2.8 is given in List 2.9.

```
<wsdl:definitions targetNamespace="http://localhost:8080/axis/Triangle.jws"
xmlns:apachesoap="http://xml.apache.org/xml-soap"
xmlns:impl="http://localhost:8080/axis/Triangle.jws"
xmlns:intf="http://localhost:8080/axis/Triangle.jws"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <types> .... </types>

 <message> ... </message>
   .....
 <portType> ... </portType>

 <binding> ... </binding>

 <service> ...</service>

</wsdl:definitions>
```

List 2.9. An Example of a WSDL *definitions* Element

2.6.4 SOAP

SOAP is a XML-based protocol that is used for exchanging structured information between heterogeneous applications in a decentralized, distributed environment (W3C, 2007).

SOAP was designed by W3C in the year 2000, in the year 2001 SOAP became the core of the XML based distributed computing (Graham, 2005).

In the Web Service architecture described in Fig. 2.3, SOAP plays the role of the messaging protocol that is used by the Service Requester and Service Provider to exchange information.

As explained in section 2.6.3, WSDL describes three types of messages: *request*, *response*, and *fault* message, SOAP is a mechanism for defining these messages using XML.

The root XML element of any SOAP message is the *Envelope* element. It consists of two elements: an optional *Header* element and a *Body* element.

The *Envelope* defines the various XML namespaces that are used by the rest of the SOAP message.

The *Header* element carries auxiliary information such as authentication, encoding or information for the intermediate recipients of the SOAP message, where a SOAP message may be received by many recipients (sometimes called nodes) until it reaches the Web Service endpoint (Service Provider) in case of *request* messages, or the Service Requester in case of *response* or *fault* messages.

The *Body* element contains information for the Service Provider or the Service Requester. The information inside the *Body* element is different depending if the message was an *input*, *output*, or a *fault* message.

To get better understanding of SOAP, a real *input*, *output*, and *fault* SOAP messages will be discussed; List 2.4 to List 2.9 were all taken from a WSDL document that describes a Web Service that provides two operations, namely: *triangleType* and

triangleArea as can be seen in the WSDL's *portType* element in List 2.5. The Service Requester can use this information inside WSDL to invoke or bind to this Web Service.

For the *TriangleService* Web Service in List 2.6:

a) *request* message

The request message is an RPC that is made by the Service Requester to obtain a certain functionality that is provided by the Service Provider of the Web Service. Each request message can include only one Web Service operation.

In order for a Service Requester to invoke the *triangleType* operation from the *TriangleService* Web Service described by the WSDL's elements in List 2.4 to List 2.9, he must extract the following information from these elements:

1. The required *operation name (triangleType operation)*, this information can be obtained from the WSDL *name* attribute of the *operation* element which is a sub-element of the *portType* element (see List 2.4, the *operation name* attribute in Fig. 2.6 and Table 2.4). The operation *name* become an element inside the SOAP *request* (see Listing 2.10).
2. The *namespace* that defines the *triangleType* operation (see *namespace* attribute and relation in Fig. 2.6 and Table 2.4)
3. The encoding style of the SOAP request to the *triangleType* operation. This information can be obtained from WSDL by first extracting the WSDL *binding* element and then extracting the *encodingStyle* attribute of the *triangleType operation* element which is a sub-element of *binding* (see List 2.5, encodingStyle attribute and relation in Fig. 2.6 and Table 2.4). In the request SOAP message to the *triangleType* operation (List 2.10) the encoding style is defined using SOAP

encoding which is available at the namespace:

"*http://schemas.xmlsoap.org/soap/encoding*"

4. The parameters for the *triangleType* operation, this information can be obtained by first knowing the *input message* to this *operation* (*triangleTypeRequest*) which can be obtained from the *message* attribute of the *input* element of the *triangleType* operation element inside the *portType* element (see input message attribute and relation in Fig. 2.6 and Table 2.4), and after that the collection of parameters to this *message* (a, b, and c) are obtained using the *part* elements' name attribute of this input message in the message element (see List 2.4, *part name* attribute in Fig 2.5 and Table 2.4). When sending a SOAP request message, the Service Requester does not use the actual parameter names, but rather the parameters or arguments to a certain operation (in this case *a, b,* and *c*) are encoded inside SOAP as *arg0*, *arg1*, and *arg2* respectively (see List 2.10)
5. The datatype of the parameters in 4 (*integerLessThanOrEqualHundred*), this information can also be obtained from the message element, as in 4, but using the *type* attribute.
6. The order of the parameters to the *triangleType* operation (*a b c*), this information can be obtained from the WSDL's *paramOrder* attribute of the *operation* element inside the *portType* element.
7. The *namespace* or URI that define the XML Schema datatypes (xmlns:xsd=http://www.w3.org/2001/XMLSchema). To ensure interoperability between Web Services, the datatype that is used in WSDL is only XML Schema datatypes.

```
POST /axis/Triangle.jws HTTP/1.0
Content-Type: text/xml; charset=utf-8
Accept: application/soap+xml, application/dime, multipart/related, text/*
Host: 127.0.0.1:8081
Content-Length: 1074

<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>

        <ns1:triangleType
        soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding"
        xmlns:ns1="http://localhost:8081/axis/Triangle.jws">
            <ns1:arg0 href="#id0"/>
            <ns1:arg1 href="#id1"/>
            <ns1:arg2 href="#id2"/>
        </ns1:triangleType>

        <multiRef id="id0"
        soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding"
        xsi:type="soapenc:int"
        xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/">51
        </multiRef>
        <multiRef id="id1"
        soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding"
        xsi:type="soapenc:int"
        xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/">50
        </multiRef>
        <multiRef id="id2"
        soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding"
        xsi:type="soapenc:int"
        xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/">54
        </multiRef>
  </soapenv:Body>
</soapenv:Envelope>
```

List 2.10. An Example of a SOAP Request with three *int* inputs (51, 50, 54)

8. The way to invoke the Web Service that has the *triangleType operation* and the transport protocol that must be used to invoke this *operation* can be obtained from the transport attribute of the WSDL's *binding* element (see List 2.4, transport attribute in Fig 2.5 and Table 2.4). In our example the transport attribute is *transport=http://schemas.xmlsoap.org/soap/http* which means that and HTTP protocol over HTTP are the transport/messaging protocols.
9. The *address* of the *TriangleService* Web Service (see List 2.6, address attribute in Fig. 2.6 and Table 2.4) that contains the *triangleType* operation (*location=http://localhost:8080/axis/Triangle.jws*). Notice that the location or address of the required Web Service does not appear in the SOAP *envelope* but rather in the HTTP request URI (see List 2.10) (POST /axis/Triangle.jws).

Using all of this information, the Service Requester can send a SOAP message as a request to the *triangleType* operation which is delivered to the Service Provider using HTTP POST method as described in List 2.10. The SOAP payload can be transported by some other HTTP methods such as HTTP GET, however, the HTTP binding defined in the SOAP specification requires the use of the POST method.

All of the information that is needed for this invocation is provided by WSDL (see Fig. 2.6), so the Service Requester needs only the information inside WSDL to make RPC to the Web Service that is described by this WSDL.

Fortunately, the Service Requester need not extract all of the previous information from WSDL in order to make a SOAP request because there are many tools or SOAP engines that can do that automatically such as Apache Axis (Apache Software Foundation, 2007).

b) *response* message

After the Service Request sends a SOAP request message to the Service (Web Service implementation) *TriangleService*, will receive a SOAP response from this Service that is listed in Listing 2.11.

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Set-Cookie: JSESSIONID=ADD05720075AD54687EAD7A22CB28BBD; Path=/axis
Content-Type: text/xml;charset=utf-8
Date: Thu, 16 Aug 2007 22:59:30 GMT

<?xml version="1.0" encoding="utf-8"?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<soapenv:Body>
<ns1:triangleTypeResponse
soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:ns1="http://localhost:8081/axis/Triangle.jws">
<triangleTypeReturn xsi:type="xsd:string">Scalene</triangleTypeReturn>
</ns1:triangleTypeResponse>
</soapenv:Body>
</soapenv:Envelope>
```

List 2.11 An Example SOAP *response* message to the SOAP *request* message in List 2.10

Like the request message in List 2.10, the response message in List 2.11 contains an HTTP header. The response code of 200 in the header is an indication that the server was able to process the SOAP payload.

The *TringleType* operation (method or function) HTTP/SOAP invocation is similar to invoking the following function Object Oriented programming languages like Java:

public String triangleType (int a, int b, int c);

This function takes three parameters a, b, and c that represents the length of the sides of a triangle and returns the type of this triangle depending on these lengths.

The SOAP request in List 2.10 invoked this operation giving the parameters 54, 51 and 50, which is similar to *triangleType (54, 51, 50)* method call in Java.

The Web Service that provides the *triangleType* operation responded in another SOAP message (List 2.11) that gave the Service Requester the type of such a triangle (*Scalence*).

Using WSDL does not only give the Service Requester what information they need to send a request to a Web Service, but also what information they should expect from this Web Service.

As noticed in List 2.11 all of the information there was already described by the WSDL document for this *TraingleService* Web Service (List 2.4 to List 2.9) these include:

1. The name of the response message (*triangleTypeResponse)* (see WSDL's *portType* element in List 2.4, output message attribute and relation in Fig 2.5 and Table 2.4)
2. The returned parameter (*triangleTypeReturn*) (see WSDL's *message* element in List 2.8, *part name* attribute in Fig 2.5 and Table 2.4)
3. The namespace associated with *triangleTypeRespone* message
4. The encoding style or serialization (marshaling) rules associated with the response message (*encodingStyle=http://schemas.xmlsoap.org/soap/encoding/*) (see List 2.5, *encodingStyle* attribute and relation in Fig. 2.6 and Table 2.4)
5. the datatype of the returned parameter (type="xsd:string")

c) *fault* message

If the request to a certain Web Service operation fails for some reason, the Service Request will receive a fault SOAP message that describes the causes of the fault and the exception handling information.

SOAP fault message in Web Services are similar to throwing an exception in Java; when a Java program throws an exception, this is an indication that something went wrong; the exception gives information on the cause of the problem. The same thing can be said in SOAP faults where the exception and its detail are sent by a normal SOAP message to the Service Requester.

To continue the *TriangleType* operation example (List 2.10 and List 2.11), a SOAP *request* message that is similar to that in List 2.10 was sent to the *TriangleService* (see List 2.6), however, this time the first parameter value, which is supposed to be an *integer* in WSDL, was replaced by a *random string* value. The Web Service responded with the SOAP *fault* message in List 2.12.

The error code 500 with the explanation "Internal Server Error" in the HTTP header indicates that a problem has occurred. The Web Service container (see Fig. 2.4) uses the error code 500 ("Internal Server Error") to tell the Service Requester that an error has occurred while processing the request message. The reason for the error or problem will be explained to the Service Requester in the *fault* element of the fault message.

There are many network-related error responses, other than "500: Server Internal Error", such as: "404: Not Found" and "Connection Timed out". Apache (Apache, 2005) discusses all of these error codes.

Since this thesis aims to assess the robustness and other related quality attributes of a Web Service, the only error code that will be considered is "500: Server Internal Error"

because it is the only one that in concerned with the Web Services implementation and the server stub implementation (middleware or SOAP engine) rather than the problems of the network between the Service Provider and the Service Requester (see Fig. 2.4).

```
HTTP/1.1 500 Internal Server Error
Server: Apache-Coyote/1.1
Set-Cookie: JSESSIONID=A658D3E32D0D73C0811926CC6815A8C2; Path=/axis
Content-Type: text/xml;charset=utf-8
Date: Thu, 06 Sep 2007 22:43:11 GMT

<?xml version="1.0" encoding="utf-8"?>
<soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <soapenv:Body>
   <soapenv:Fault>
       <faultcode>soapenv:Server.userException</faultcode>
       <faultstring>
          org.xml.sax.SAXException: Bad types (class java.lang.String -&gt; int)
       </faultstring>
       <detail>
         <ns1:hostname xmlns:ns1="http://xml.apache.org/axis/">
              e-sci030
         </ns1:hostname>
       </detail>
   </soapenv:Fault>
 </soapenv:Body>
</soapenv:Envelope>
```

List 2.12 An Example SOAP *fault* message

According to the SOAP specification if the request message is received and understood, the respond should be sent by the 200 status code. In case that the server does not understand the message, or the message format is wrong such as missing information, or the message can not be processed for any other reason, the server must use HTTP code 500 (Englander, 2002).

The *body* element of a SOAP *fault* message contains a *fault* element; this element is responsible for the explanation to the Service Request on what has gone wrong. To achieve this, fault element has many sub-elements or components that give a description of the fault or error occurred, these elements include:

- ***faultcode***, this component describes in general what the problem was, there are four codes to describe what type of fault occurred:
    1. *Server*: This code means that something went wrong when the receiver tried to process the request message, where the receiver could be: Web Service implementation, Web Service container or server stub (see Fig. 2.7).
    2. *Client*: This fault code means that there was something incorrect in the request SOAP message such as missing data. In other words, the request message was incorrectly formed.
    3. *VersionMismatch* (Graham, et al. 2005).
    4. *MustUnderstand* (Graham, et al. 2005).

    VersionMismatch and Mustunderstand are not related to the research line in the thesis.

    In the SOAP fault message of List 2.12, the *faultcode* is *Server.userException* which means that the fault is generated by the server side (see Fig. 2.4) because the server stub raised an exception since the request (from the user of the Web Service or Service Requester) has wrong datatype which is string and not integer as described by WSDL.

    In other words, the request did not reach the Web Service implementation because it was intercepted by the server stub or skeleton.

- ***faultstring***, this element contains a human-readable description or explanation of the fault.

  In List 2.12 the faultstring is org.xml.sax.SAXException: Bad types (class java.lang.String -> int) and there are two notes about this faultstring:

  1. *The "&" character is escape character in XML to replace the "<" and ">" signs because obviously they have special meaning in XML which is surrounding the elements names. "&gt" stands for the ">"symbol.*
  2. *The exception is a Simple API for XML (SAX) (Harold, et al. 2004) parser exception because the new versions of Apache Axis uses SAX rather Document Object Model (DOM) (W3C, 2005) parser that was used in earlier versions. Obviously the server stub or skeleton in this example was built using Axis.*

- ***faultactor***, before a request SOAP message reaches its destination (Web Service implementation) it may pass through intermediate nodes or entities on its way; *faultactor* element specifies which entity of these caused the fault.

  In List 2.12 this element does not exist because the fault happened in the final destination of the message (Web Service container side in Fig. 2.4).

detail, this element provides more information about the fault (other than fault code and fault string) such as a stack trace of the fault which is considered an application specific information.

In List 2.10 the *detail* element only gave the name of the server that contains the Web Service container of the targeted Web service implementation.

In summary, the SOAP fault message carries to the Service Requester all the information he needs to know why a fault has occurred to help in sending a correct SOAP request next time.

2.6.5 UDDI

Universal Description, Discovery, and Integration (UDDI) is a standard plays the role of the broker or registry in SOA (Fig 2.1). This standard helps the Service Requester to discover or locate Service Providers and retrieve a description of the Web services they provide.

A UDDI Registry provides information about published Web Service and their Service Providers such as:

- The address and contact of the Service Provider of the Web Service
- Where the Web Service can be accessed (URL).
- A short description of what the Web Service does
- Technical information of how to bind to the Web Service.
- The location of the WSDL document

After the Service Requester retrieves the WSDL document they can use the information there to invoke the described Web Service implementation as discussed in Section 2.5.

A repository of WSDL document can also play the role of the Service Registry in SOA (Graham, et al. 2005). An example of a public repository of WSDL documents is XMethods (http://www.xmethods.com).

The WSDL repository is simpler than using UDDI, however, UDDI is more dynamic because it enables Service Requesters to search, find, and bind to the required Web Service at run time.

2.7 Summary

This chapter gave a definition of SOA and discussed the characteristics of the Service in a SOA. The Web Service architecture was discussed then in order to explain how Web Service impalement SOA. Different definition of Web Services was then surveyed and a new definition that includes all Web Services characteristics was introduced. After that the components that participate in a Web Service invocation were discussed. The Web Services open standards was then discussed with more details to the standards that are of more importance to the Web Services testing approach that is developed in this thesis.

Chapter 3

Software Testing and Quality Attributes

3.1 Introduction

This chapter will survey software quality attributes with more details about robustness and the other quality attributes that are related to this thesis approach.

Since testing techniques are used to assess quality attributes, this chapter introduces a survey on the software testing techniques that are related to this thesis. Finally, a survey on the available robustness testing tools will be introduced.

3.2 Quality Attributes

Quality attributes are the key factors in the success of any software system. Also quality attributes are important for the user of the software system to evaluate how good a system is. However, software quality is a complex and subjective mixture of several attributes or factors and there is no universal definition or a unique metric to quantify software quality (Raghavan, 2002).

Software quality is measured by analyzing the various attributes that are significant to a certain domain or application (Raghavan, 2002). According to Garvin (Garvin, 1984) quality can be described from five different perspectives. One of these is the user view. A user sees quality as "fitness of purpose", i.e., quality is defined as the product characteristics that meet the user needs or expectations whether explicit or not.

The quality attributes literature includes the following main quality models: Boehm (Boehm, 1976), McCall (McCall, 1977), Adrion (Adrion et al. 1982), and ISO 9126:2001 (ISO 9126-1, 2001).

When analyzing the main quality models, it is noticed that there is no agreement between researchers about a fixed general quality attributes because there is no shared understanding about the quality attributes (or characteristics). For example, the terms accuracy and correctness are used by different researchers to mean the same quality attribute. Also it is noticed that some sub-attributes are related to different attributes. For example: accuracy is related to the functionality attribute in ISO 9126, while it is related to reliability attribute in Boehm's model; and, although being mainly related to security, access control is related to integrity in McCall's model.

The software attribute that is of interest to this thesis is *trustworthiness*.

Trustworthiness is defined as:

> "*Assurance that the system will perform as expected*". (Avizienis et al., 2004).

Another definition of trustworthiness is that it is:

> "*Well-founded assessment of the extent to which a given system, network, or component will satisfy its specified requirements, and particularly those requirements that are critical to an enterprise, mission, system, network, or other entity*" (Neumann, 2004).

Some quality attributes have sub-attributes which are considered as requirements for the main attribute (see Fig. 3.1); trustworthiness requires many quality attribute such as: security, *reliability, safety, survivability, interoperability, availability, fault tolerance*, and *robustness*, etc. (Zhang, J., 2005c). However, fault-tolerance and robustness are sub-attributes of reliability (ISO 9126-1, 2001) (Adrion, 1982); Fig 3.1 describes the

trustworthiness quality model according to these relations between the quality attributes.

The trustworthiness attribute needs some sub-attributes. And these sub-attributes themselves have sub-attributes, as shown in Fig. 3.1.

To assess the trustworthiness of any software system, researchers and practitioners must find methods to assess the trustworthiness sub-attributes such as reliability, security, and so on.

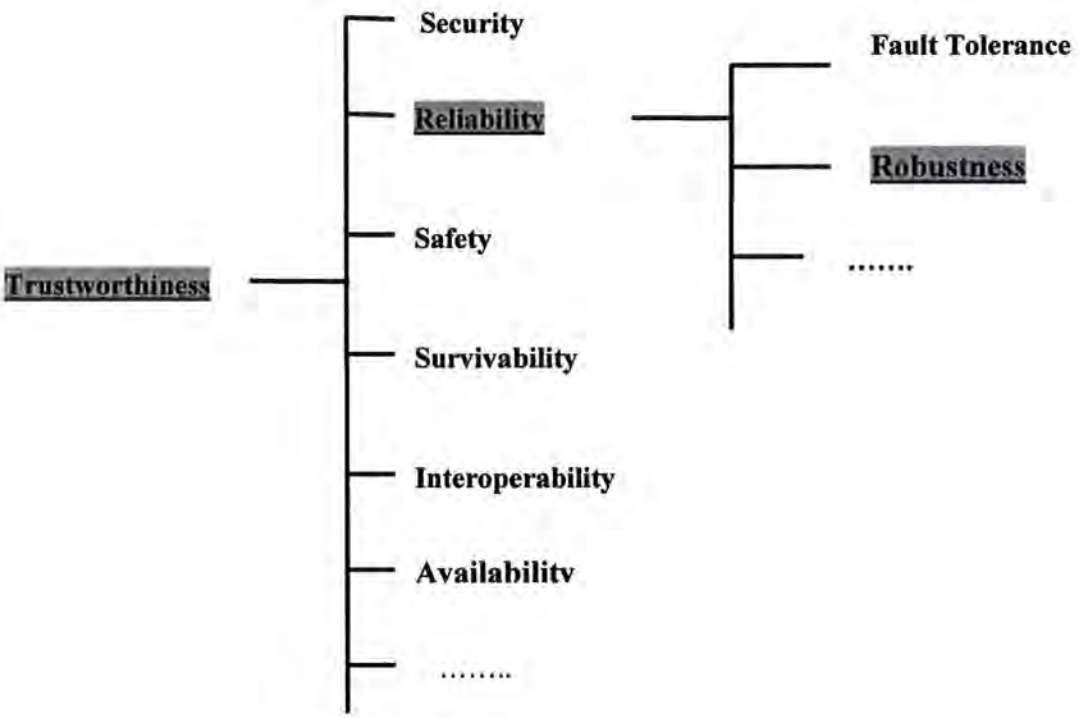


Fig. 3.1. Trustworthiness Quality Model

Avizienis (Avizienis et al., 2004) stated that *dependability* and trustworthiness have the same goals and they both face the same threats (faults, errors, and failures).

Dependability is defined as

> "*Ability to deliver a Service that can justifiably be trusted" or "ability of a system to avoid Service failures that are more frequent or more severe than is acceptable*" (Avizienis, 2004).

Dependability encompasses the following sub-attributes: *availability, reliability, safety, integrity, maintainability.* (Avizienis, 2004).

In one piece of research, it is very difficult to discuss all the trustworthiness and dependability related quality attributes such as reliability, security, etc. This thesis is concerned mainly with the reliability quality attribute. (Discussing and assessing other trustworthiness attribute will be left as future research).

Reliability is defined as:

> "*Ability to tolerate various severe conditions and perform intended function*" (Raghavan, 2002).

Another similar definition of reliability is that it is:

> "*Requirements might include properties relating to the ability to tolerate hardware failures and software flaws, the characterization of acceptable degradation in the face of untolerated faults, probabilities of success, expected mean times between failures, and so on. Measures of reliability typically represent the extent to which flaws, failures, and errors can be avoided or tolerated*" (Neumann, 2004).

Another definition of reliability is:

> *"The probability that software will not cause the failure of a system for a specified time under specified conditions"* (IEEE, 1990).

The first definition by Reghavan implies that reliability is related to fault-tolerance and robustness (tolerate severe conditions), also reliability is related to correctness in this definition (perform intended function)

The second definition by Neumann implies that reliability is related to fault-tolerance (tolerate hardware failures and software flaws), robustness (the extent to which flaws, failures, and errors can be avoided), correctness (probabilities of success), and it also introduces mean time between failures as a measure of reliability.

The third definition by IEEE implies that reliability is related to robustness and fault tolerance.

Some researchers such as Adrion (Adrion et al. 1982) discussed the reliability requirements, and state that reliability requires the following sub attributes: correctness, completeness, consistence, robustness, maturity, fault-tolerance, and recoverability.

To assess how reliable a software system is, these entire requirement (or sub-attributes) of reliability must be assessed.

As it is difficult in a single piece of work to assess the entire trustworthiness requirement, this thesis mainly focuses on the robustness sub-attribute of reliability.

To achieve robustness and fault tolerance; robustness testing and other fault-based testing techniques are required (see section 3.4 and 3.5)

Robustness quality attribute is defined as:

> *"The degree to which a system or component can function correctly in the presence of invalids input or stressful environmental conditions"* (IEEE, 1990)

While **fault-tolerance** quality attribute is defined as:

"The ability of a program to produce acceptable output, regardless of what potential problem arise during execution" (Voas, 1996).

Security is defined as the quality attribute that defines confidentiality for parties using software (Looker, et al., 2007).

3.3 Testing Definitions

The testing literature is mired with confusing and inconsistent terminology because it has evolved over decades and by different writers (Jorgensen, 2002). This section will introduce a definition of testing and the related terms that will be used through this thesis.

The testing literature has the following main definitions of testing:

1. IEEE (IEEE, 1990)

   "An activity in which a system or component is executed under specified conditions, the results are observed or recorded, and evaluation is made of some aspect of the system or components".

2. Hetzel (Hetzel, 1973)

   "The process establishing confidence that a program or system does what it suppose to".

3. Myers (Myers, 1979)

   "The process of executing a program or system with the intent of finding errors".

4. Beizer (Beizer, 1990)

   "A process that is part of quality assurance and aims to show that a program has bugs (faults)".

5. Voas (Voas and McGraw, 1998a)

"the process of determining whether software meets its defined requirements".

6. Harrold (Harrold, 2000)

   "One of the old forms or verification that is performed to support quality assurance".

7. Sommerville (Sommerville, 2004) defines testing as:

   "Software testing involves running an implementation of the software with test data. You examine the outputs of the software and its operational behavior to check that it is performing as required. Testing is a dynamic technique of verification and validation."

These definitions introduce testing-related terms such as quality assurance, fault, error, verification and validation.

The goal of quality assurance is to improve software quality and to determine the degree to which the actual behavior of the software is consistent with the intended behavior or quality of this software. Quality assurance activities may include: inspections, reviews, testing, and audit (Raghavan, 2002). However, this thesis concerned with increasing the quality assurance using testing only.

The following terms are defined to enable a better understand of testing definitions:

1) Fault, Errors, and Faults:

Fault, error, and failure are considered as a threat to the dependability (see section 3.4) of a system (Avizienis et al. 2004) and they are defined as follows:

Fault is defined as:

"*A defect in the system that may lead to an error*" (Osterweil, 1996); another name of a fault is bug or defect.

IEEE (IEEE, 1995) presented a comprehensive treatment or classification of the types of faults that may affect a software system such as input faults, output faults, and computation faults. Avizienis (Avizienis et al. 2004) classified faults to fault classes such as malicious and non-malicious faults, internal and external faults.

For a certain quality attribute there exist faults that affect this quality attribute. Examples of faults that may affect *robustness* quality attribute include: *wrong input accepted, correct input rejected* (IEEE, 1995). Some faults can affect more than one quality attribute, for example, *wrong input accepted* fault affects *robustness, fault tolerance and security.*

Error is defined as

"The part of the total state of the system that may lead to a failure" (Avizienis et al. 2004).

Failure is defined as

"the deviation of the execution of a program from its intended behavior" (Osterweil, 1996)

Another definition of failure is:

"*An event that occurs when the delivered service deviates from correct service"* (Avizienis et al. 2004).

Avizienis (Avizienis et al. 2004) also stated that:

*"The prior presence of a **vulnerability**, i.e., an internal fault that enables an external fault to harm the system, is necessary for an external fault to cause an error and possibly subsequent failure(s)"*

So fault may lead or cause an error, which consequently may lead to a failure when it reaches the system's external state.

2) Verification and Validation

Verification and validation (V & V) is the process of checking that a program meets its specification and delivers the functionality expected by the people paying for the software (Sommerville, 2004). Verification and validation are defined as follows:

Verification is defined as:

"Checking that the software conforms to its specification and meets its specified functional and non-functional requirements" (Sommerville, 2004)

Validation is defined as

"Ensure that the software system meets the customer's expectations" (Sommerville, 2004)

Another definition of validation is

"Determination of the correctness of the final program or software produced from a development project with respect to the user needs and requirements" (Adrion , et al. 1982).

After defining testing and the related terms, this thesis will return to the different definitions of software testing to extract the roles of software testing in these definitions; it is noticed that different researchers view software testing differently,

however, the following roles or goals of software testing can be included from the definitions:

1. Testing involves running or executing the system under test with test data.
2. Testing is a performed to support quality assurance by assessing the quality attributes
3. Testing is performed to find faults before they cause an error and consequently a failure
4. Testing is a form of verification.
5. Testing is a form of validation.

However, these testing roles overlap with each other because:

- Faults are related to quality attributes; by finding a fault we are actually assessing the quality attributes or attributes that are related to this fault.
- Verification and validation includes assessing quality attribute and accordingly supporting quality assurance.
- Finding faults that may lead to errors and failures is considered part of verification and validation.

Table 3.1 analyzes the roles in each definition of software testing in order to reach a definition that contains all the testing roles. The table indicates whether we can infer a role (column) based on a particular definition (row).

The symbols shown in the table are:

1. The full circle (●) indicates that the definition explicitly states the role.
2. The symbol (≈) indicates that the definition does not explicitly express that specific role, but the context of the definition suggests it.
3. The empty circle (○) indicates that the role is not included in a specific definition.

Table 3.1. Relations between Software Testing Definitions and Roles

Role / Definition	Executing system with test data	Assessing quality attributes	Finding faults	Verification	Validation
IEEE (IEEE, 1990)	●	≈	○	○	○
Hetzel (Hetzel, 1973)	○	○	○	○	≈
Myers (Myers, 1979)	●	○	●	○	○
Beizer (Beizer, 1990)	○	●	●	○	○
Voas (Voas and McGraw, 1998a)	○	○	○	●	○
Harrold (Harrold, 2000)	○	●	○	●	○
Sommerville (Sommerville, 2004)	●	○	○	●	●

It is noticed from table 3.1 that Sommerville (Sommerville, 2004) definition contains more of the software testing roles than the other definitions.

After analyzing all the definitions, this thesis will use the following definition of software testing that includes all the roles mentioned in Table 3.1:

> *Software testing is a quality assurance process that is part of the verification and validation processes, and involves executing the system under test with test data for the purpose of detecting faults and assessing the quality attributes of that system or software component.*

3.4 Testing Techniques

Testing consists of the following steps (Harrold, 2000):

1. Designing test data
2. Executing the system under test with those test cases
3. Examining the results of the execution and comparing them with the expected results.

This means that the program to be tested is executed using representative data samples or test data and the results are compared with the expected results.

Test cases include input test data and the expected output for each input. It is impossible to test a piece of code, such as a method or function, with every possible input to check if the code produces the expected output. This is known as exhaustive testing (Voas and McGraw, 1998a). However, there are many testing techniques that are used to design test data such as *boundary value testing* and *equivalent partitioning*.

Testing techniques can be categorized along various dimensions depending on:

- **The availability of the source code**

  Testing techniques can be categorized to black-box or white-box testing according to the availability of the source code:

  White-Box testing

  If the source code of the system under test is available then the test data is based on the structure of this source code (Jorgensen, 2002).

  Examples of white-box testing are: path testing and data flow testing (Jorgensen, 2002).

Black-Box testing

If the source code is not available then test data is based on the function of the software without regard to how it was implemented (Jorgensen, 2002). Examples of black-box testing are: boundary value testing (Jorgensen, 2002) and equivalence partitioning (Myers, 1979).

- **The role of testing**

Testing techniques can also be categorized according to the type of testing (Sommerville, 2004) which is based on the role or goal of this test; some testing techniques belong to the validation testing and others belong to the defect or fault-based testing:

Validation testing

This kind of testing is intended to show that the software meets the customer requirement. In validation testing each requirement must be tested by at least one test case.

An example of a testing technique that belong to this type of testing is specification-based testing (Offutt et al. 1999) (Offutt et al. 2003) where test data are generated from state-based specifications that describes what functions the software is supposed to provide.

If the specification is written by a model such as UML and the test case generation is based on that model, then the testing is called model-based testing (Toth et al. 2003), This testing also belong to validation testing.

Correctness and *accuracy* (see section 3.4) are examples of the quality attribute that can be assessed by validation testing.

Defect testing (fault-based testing or negative testing)

This type of testing is intended to detect faults (bugs or defects) in the software system rather than testing the functional use of the system like validation testing (Sommerville, 2004).

Examples of the testing techniques that belong to this type of testing include: fault injection (Voas and McGraw, 1998a), boundary value based robustness testing (Jargensen, 2002), and syntax testing (Beizer, 1990).

Defect testing contribute to the assessment of the following quality attributes: robustness, fault-tolerance and security (see section 3.4)

Since fault-based testing is the type of testing that is used in this thesis' method of testing Web Services, then it will be discussed, with more details, in an independent section (section 3.5).

- **The level of testing**

Testing techniques can be distinguished according to the scope or level of a test:

Unit testing

Testing individual or independent software unit (IEEE, 1990). A unit is defined as the smallest piece of software that can be independently tested (Beizer, 1990).

Integration testing

This kind of testing is used to test the interaction between the units that was already tested using Unit testing (IEEE, 1990).

System testing

This kind of testing is conducted on a complete and integrated software system to evaluate its compliance with its specified requirements (IEEE, 1999).

- **The quality attribute or system behavior**

Testing techniques can be distinguished according to the quality attribute or system behavior being tested such as performance, robustness, and correctness. Examples of these kinds of testing are:

Performance testing

Used to assess the performance quality attribute of a system or component and is defined as:

"Testing conducted to evaluate the compliance of a system or components with specified performance requirements" (IEEE, 1990).

A performance requirement may include speed with which a given function must be performed (IEEE, 1990).

Robustness testing

Robustness testing is used to assess the robustness quality attribute of a software system. Robustness testing include some testing techniques such as boundary value based robustness testing technique (Jorgensen, 2002) and Interface Propagation Analysis (IPA), both of these techniques will be discussed in section 3.5.

Robustness testing is defined as:

Testing how a system or software component reacts when the environment shows unexpected behavior (Dix and Hofmann, 2002).

Security testing

Used to assess the security quality attribute of a system or component by testing if an intruder can read or modify the system data or functionality.

Load testing

Used to test if a system or component can cope with heavy loads such as being used by many users at the same time.

Regression testing

Regression testing is defined as

"A form of black box testing in which a component's functionality is compared to the functionality of a previous version of that component, to verify that changes to the component haven't broken anything that worked previously". (Bloomberg, 2002)

Although assessing quality attributes belongs to validation testing, for some quality attributes, such as robustness, we must analyze the faults that affect this quality attribute to be able to test if a software system has such faults. However, other quality attributes or system behavior such as performance does not need such fault analysis.

The above four categories or dimensions (the availability of source code, the role of testing, the level of testing, and the quality attribute or system behaviour) of testing techniques are not disjoint; for example the boundary value based robustness testing (Jorgensen, 2002) belongs to the following types of testing: black-box testing, unit testing and fault-based testing at the same time. Other black-box testing techniques can also be considered fault-based testing techniques such as syntax testing (Beizer, 1990).

3.5 Fault-based Testing

Fault-based or negative testing is defined as

"Testing aimed at showing software does not work" (Beizer, 1990)

Testing that the system meets its requirement (validation testing) without applying fault-based testing leave the software system open to vulnerabilities that might not surface until much later in the development cycle or after deployment (Cohen, et al. 2005)

Fault based testing aims to solve this problem by discovering the following (Lyndsay, 2003):

- Faults that may result in significant failures
- Crashes
- Security breaches
- Observation of a system's response to external problems
- Exposure of software weakness and potential of exploitation

Fault-based testing is important because even though a software component has been tested using unit testing and some black-box testing techniques, this does not mean that this component of high quality because we must check if this component has vulnerabilities to faulty input.

In fault-based testing, test cases are written for invalid and unexpected input conditions in order to check how if the system under test will can handle such input gracefully.

Handling an invalid input gracefully may include raising an exception with a proper error message that describes to the user what happened, while if the system has vulnerabilities to such invalid inputs, then it might reveal important information that can be used by malicious used to harm the system.

Systems that have an interface which is accessible by public must specially be robust and consequently must have prolific input-validation checks (Beizer, 2002).

Myers (Myers, 1979) states that test cases which contain invalid and unexpected input conditions seem to have higher error or fault detection rate than do test cases for valid and expected input conditions.

The fault-based testing techniques that are important to the research in this thesis are: Interface Propagation Analysis (IPA) which is one of the fault injection techniques (Voas and McGraw, 1998a) (Voas, 1998b), robustness testing (Jorgensen, 2002) and syntax testing (Beizer, 1990) which belong to black-box testing techniques.

3.5.1 Fault injection

Fault injection includes a group of techniques that are important to evaluating the dependability of computer systems (Hsueh, et al. 1997).

Fault injection can be used with hardware or software. This thesis is concerned only with software-level fault injection.

Most of the fault injection techniques belong to white-box testing because they require injecting faults to the source code to assess its fault-tolerance. An example of the fault injection techniques is the *mutation testing* (Osterweil, 1996) which is the process of "re-writing" source code by making a small change in the code to produce what is called a mutant. A test execution that demonstrates such difference is said to kill

the mutant. This is done to flush out ambiguities or vulnerabilities that may exist in the code. These ambiguities can cause failures in software if not detected and fixed.

3.5.1.1 Interface Propagation Analysis (IPA)

IPA is defined as:

"*A fault-injection based technique for injecting 'garbage' into the interfaces between components and then observing how that garbage propagates through the system*" (Voas, 1997).

IPA predicts how software will behave when corrupt information get passed (Voas et al. 1996). IPA assess if problems may enter the component based systems from its environment when this environment behaves unexpectedly by sending corrupted data to a component. IPA offers an approach to assessing the robustness of systems based on COTS components (Voas and McGraw, 1998)

3.5.2 Boundary Value Based Robustness Testing

This testing technique is an extension to boundary value testing (Jorgensen, 2002). The test cases include the values at the boundary of the input parameters (as boundary value testing) and also the value above the maximum value and below the minimum value of this parameter.

It is expected that the system under test will produce a proper error message when the input to this system exceed its boundaries. The main advantage of boundary value based robustness testing is that it forces attention on exception handling (Jorgensen, 2002).

3.5.3 Syntax Testing with Invalid Input

Syntax testing is an input data validation testing technique that is used to test the system's tolerance for bad data (Beizer, 1990). Test cases are based on a formal description of the input parameters that is understood by the interface of software. An example of formal description is when the input parameters are described using regular expression.

Beizer (Beizer, 2002) described different kinds of errors that can be generated using syntax testing such as:

Syntax errors

These kind or errors are generated by violating the grammar of the specification language. An example of such errors includes: remove last character, replace last character, add extra character, and remove first character (Murnane et al. 2006).

Delimiter errors

Delimiters used to separate the fields on an input; an example of a delimiter is space or dash. Delimiter errors may include omitting the delimiter, replacing it with different delimiter.

3.5.4 Equivalence Partitioning with Invalid Equivalence Classes

Equivalence partitioning testing techniques include partitioning the input space or domain into a finite number of equivalence classes that include a specified set of input values (Myers, 1979). Each member of an equivalence class is supposed to make the

system under test behave the same and so we only have to use one member of the class for test data.

Equivalence classes may be valid or invalid, however, since the fault-based testing is important to this thesis, only invalid equivalence classes will be considered.

Equivalence partitioning technique does not clearly define how to select invalid test data because the invalid data may include all inputs other than those specified as valid (Murnane, 2005). Murnane (Murnane, 2005) suggested some invalid equivalent class such as: integer replacement, real replacement, and null replacement. Table 3.2 summarizes the different fault based testing techniques described in section 3.5.

Table 3.2. Test Data Generation Method in Fault-based Testing Techniques

Testing Technique	Test Data Derivation Method
IPA	Feeding a software component a "Garbage" input
Boundary value based robustness testing	Choose test data around the boundaries of the input parameter
Syntax testing with invalid input	Violate the rules of the specification of the input parameter
Equivalent partitioning with invalid partition class	partitioning the input space or domain into a finite number of equivalence classes

The testing techniques in Table 3.2 share the following characteristics:

- Sending invalid (corrupted, faulty, erroneous, manipulated, perturbed, or garbage) input to a software component to check if this resulted in a failure
- Fault injection based testing techniques
- Black-box testing techniques
- Unit testing

- Assessing the robustness quality attribute

These testing technique also share the same failures modes which include:

- The system under test does not recognize a good input
- The system under test accepts a invalid input without raising a proper exception
- The system crashes after attempting to process invalid input

If any of these failures occurred then the system under test must be debugged in order to handle such invalid input and increase its robustness and fault-tolerance to invalid input.

3.6 Prior Work on Robustness Testing

Cohen (Cohen, et al. 2005) stated that "very limited or no testing was performed to ensure that the system could handle unexpected user input", this means that very little researche exists for assessing the robustness quality attribute because most the research on the field of software testing and quality attributes focus on validation testing rather than fault-based testing.

However, there are some research projects and associated tools that aim to assess the robustness of software systems, among these projects and associated tools: Fuzz, Ballista, RIDDLE, JCrasher, and CORBA middleware robustness testing tool; these tools are discussed in the following sections.

3.6.1 Fuzz

Fuzz (Miller, et al. 1990) is considered one of the first noted research studies on the robustness quality attribute (Schmid & Hill, 1999). The Fuzz research project was performed by a group at the University of Wisconsin in the USA; this group developed a tool that is called Fuzz. This tool depends on *random black-box testing* techniques (Jorgensen, 2002) to assess the robustness of the UNIX operating system (Miller, et al. 1990).

Although random testing is not a good testing technique in detecting faults, the research group had found that 25-33% of standard UNIX utilities crashed or hung when testing using Fuzz (Miller, et al. 1990).

3.6.2 Ballista

Ballista (Koopman, et al. 1997) is a research project that was carried out by a group at Carnegie Mellon University in the USA. This group developed the Ballista tool that is used to automatically assess the robustness of the commercial off-the-shelf (COTS) components.

A robustness failure in Ballista occurs when a component fails to handle an input that contains a combination of valid and invalid data (Koopman, et al. 1997). Automating robustness testing enables the testers to run a large number of potentially interesting tests with little interaction (Dix & Hofmann, 2002). Ballista was able to find robustness failures in components used in several commercial UNIX based Operating system (Gosh, 1998).

Unlike Fuzz which generates the test data randomly, Ballista depends on analyzing the data types of the input parameters to generate the test data (Shelton, 2000). Ballista was extended to test any component based systems and not only the operating systems components. Pan (Pan et al., 2001) extended Ballista to be used with CORBA ORB implementations.

3.6.3 RIDDLE

The Random and Intelligent Data Design Library Environment (RIDDLE) has many similarities to Ballista and both developed by same group; however Riddle is an environment that was created for testing the robustness of COTS software on Windows NT systems (Gosh, et al. 1998) rather than UNIX components.

RIDDLE uses black box testing techniques and generates anomalous input for the component under test based on this component interface specification.

Three types of input generated in RIDDLE (Gosh, et al. 1998):

- Random input
- Intelligent input based on the input grammar of the component under test that can be extracted from the specification.
- Malicious input

Generating syntactically correct but anomalous test data based on the input grammar will result in exercising more of a program's functionality than random testing (Gosh, et al. 1998).

The robustness failure modes or classes in RIDDLE include the following (Gosh, et al. 1998):

- Incorrect exit codes
- Unhandled exceptions
- Hung processes
- System crashes

3.6.4 JCrasher

JCrasher is an automatic robustness testing tool for java code (Csallner & Smaragdakis, 2000). JCrasher automatically generates random data depending on the datatype of the input parameters to the methods.

The target of JCrasher is to attempts to detect faults that cause a program to "crash", that is to throw an undeclared runtime exception (Csallner & Smaragdakis, 2000).

3.6.5 CORBA Middleware Robustness Testing

Pan (Pan et al. 2001) discussed how to assess the robustness of the ORB implementation of Common Object Request Broker Architecture (CORBA) middleware (Object Management Group, 1998).

Pan (Pan et al. 2001) stated that methods for evaluating the robustness of CORBA ORB are rare and there is an urgent need for a method to evaluate the robustness of ORB implementations.

This research uses Ballista tool to assess how graceful C++ ORB implementations handles expected and unexpected exceptions and it has found that these implementations have significant robustness vulnerabilities.

The robustness failure modes in this research are the following (Pan et al. 2001):

- Computer crash (Catastrophic failure)
- Thread hang (Restart failure)
- Thread abort (Abort failure)
- Raise unknown exception
- False success (Silent failure)
- Misleading error information (Hindering failure)

While the robust or graceful behavior include successfully return (no exception) or raise CORBA exception.

Mardsen (Marsden et al. 2002) used fault injection techniques to assess the dependability of CORBA systems.

Table 3.3 will give a comparison of the robustness testing tools according to the testing technique or test data generation method used and the platform or system targeted by each tool.

Table 3.3. Comparison of Robustness Testing Tools

Tool	Testing Technique(s)	Targeted Software System
Fuzz	Random black box testing	UNIX OS
Ballista	Automatic random black box testing	COTS of UNIX OS
Riddle	Random black box testing Test data based on input grammar Test data based on malicious input	COTS of Windows NT OS
JCrasher	Random test data based on the input datatype	Java code
CORBA Robustness Testing	Automatic random black box testing	C++ CORBA ORB implementation

The best tool is Riddle because it depends on more that one test data generation method. However, all the tools depend on random testing which is considered inefficient testing technique. The proof of concept tool of this thesis that is presented in Chapter 6 is different from these tools because it uses different testing techniques for different faults and is based on test cases rules that were systematically generated using: Web Services Description Language (WSDL), fault-based testing techniques, and the faults that may affect the robustness quality attribute of Web Services.

Software robustness testing in this thesis refers to the process of assessing the ability of software to handle invalid inputs or stressful conditions.

3.7 Summary

This chapter discussed quality attributes and software testing techniques. Quality attributes are the key factors in the success of any software system. Trustworthiness includes many sub attribute or requirements such as reliability, security, availability, and so on. Reliability itself requires robustness, fault tolerance, correctness, and other attributes.

To increase the trustworthiness of Web Services, this thesis concerns with assessing and increasing the robustness quality attribute. A definition of the trustworthiness and the related attributes was given in this chapter, also a definition of testing and testing techniques were introduced with more details about fault based testing technique because they are important in assessing robustness.

The robustness research and tools are very limited because research is usually aim at making sure that a software component or system meets its specification rather than

assessing the robustness quality attribute that try to find if a system has any vulnerabilities to invalid or faulty inputs. There exist however some robustness testing tools such as Ballista and Fuzz. Also some few researches assessed the robustness of middleware implementation such as (Pan et al. 2001).

Chapter 4

Web Services Testing

4.1 Introduction

Quality of Service is the dominant success criteria in Web Services because it is the main issue that contributes to the reluctance to use Web Services. Testing is used in this thesis to assess robustness and other related quality attributes of Web Services in order to increase Web Services trustworthiness.

Before discussing the proposed Web Service testing framework in chapter 5, this chapter will introduce the following

- A survey on the quality attributes of Web Services (section 4.2)
- A survey on the testing techniques used so far, by researchers and practitioners, to test Web Services (section 4.3)

4.2 Web Services Quality Attributes

Although quality attributes of interest may vary between Web Services applications according to the domain where they are used, we analyze and focus our work on the general abstract quality attributes that affect most of the Service Requesters of Web Services.

Zhang (Zhang, 2004) stated that Web Services trustworthiness is hindering the adoption of Web Services. Web Services trustworthiness according to this research represents people's confidence in using Web Services. The quality attributes that affects

trustworthiness according to Zhang are the same classical software attributes such as reliability, scalability, efficiency, security, usability, adaptability, maintainability, availability, portability. In particular (Zhang, 2005a) states that trustworthiness includes: security, reliability, safety, survivability, interoperability, availability, and fault tolerance.

Zhang and Zhang (Zhang and Zhang, 2005c) stated that we need to investigate how to quantitatively and qualitatively define the quality of Web services. They mentioned the same quality attribute of trustworthiness as Zhang (Zhang, 2005a) but added the testability quality attribute. It should be noted that the trustworthiness requirements or sub-attributes are different even in the researches of the same author(s).

Looker (Looker, et al. 2004) stated that the non-functional quality attributes for Web Services include: availability, accessibility, integrity, security, performance (latency and response time), reliability, and regulatory.

Some researchers are interested in a single quality attribute of Web Services such as the reliability attribute (Zhang & Zhang, 2005b) and the robustness attribute (Fu, et al. 2004). However, Zhang and Zhang (Zhang & Zhang, 2005b) stated that reliability of Web Services can be defined as a combination of six attributes: correctness (C), fault tolerance (F), testability (T), interoperability (I), availability (A), and performance (P). In other words, the reliability of Web Services will be a function of the specific six attributes:

$$R(WS) = f(aC, bF, cT, dI, eA, fP)$$

where a, b, c, d, e, and f are quantitative and qualitative measure of particular attribute. However they only considered correctness and fault tolerance in their research.

The Web Services quality attribute that is important to this thesis is robustness and it is defined by the author as:

"Web Services Robustness: the quality aspect of whether a Web Service continues to perform despite some violations of the constraints in its specification".

4.3 Web Services Testing

Web Services testing has many advantages such as increasing the trustworthiness, however, it still faces many difficulties or challenges as discussed in Chapter 1. Testing takes a whole new dimension in Web Services because applications may be composed dynamically from different available Web Services that may be located in different places and have different quality attributes. How do we test Web Services that can come from different Service Providers, hosted in different environments? Not only the source code of the Service is unavailable, the Service might be hosted on servers at remote, even competing organizations (Offut & Xu, 2004).

Current methods and technology simple cannot ensure trustworthiness in Web Services (Zhang, 2005a). Testing Web Services can be viewed from two perspectives: the Service Provider and the Service Requester. One difference between the two perspectives is the availability of the Service's source code: the Service Provider has access to the source code, whereas the Requester typically does not. The lack of source code for the consumer of the Service limits the testing that he can perform.

The Service provider should build quality into the Service in the early stages of the development of that Service and not wait until implementation to complete and then apply testing and analysis of the end Service to assure quality.

Bloomberg (Bloomberg, 2002) stated that Web Services testing tools employ the following range of traditional software testing techniques: black box (functional testing), white box (structural testing), regression testing, load testing, unit testing, and system testing. However, according to Bloomberg (Bloomberg, 2002) the traditional techniques are not able to cover the new testing issues that arise in Web Services. The desirable Web Services testing capabilities are:

- Testing SOAP messages – using SOAP to supply test cases since Web Services have no user interfaces, and also testing the format and the intermediaries of a message.
- Testing WSDL files and using them for test plan generation – using the information in WSDL files to generate black box test plans.
- Web Services consumer and producer emulation – emulating the consumer of a Web Service by sending test messages to another Web Service and analyzing the results in turn emulating the provider of the Web Service by returning a response message to the other Web Service after the consumer sends a request message.
- Testing the publish, find, and bind capabilities of an SOA
- Web Services orchestration testing – testing the composition of Web Services from other Web Services.
- Service-level agreement (SLA) and Quality of Service (QoS) monitoring – Web Services testing tools that verify at run time that Web Services are performing the way they should.

Since the robustness and other related quality attributes, such as security and fault tolerance, are important to this thesis, Table 4.1 gives a summary about research that assess robustness and other related quality attributes using fault-based testing

techniques. Table 4.2, on the other hand, will give a short survey on the researches on Web Services testing that do not use fault-based techniques.

The following issues can be concluded from Table 4.1 and Table 4.2:

- Some researchers such as (Offutt & Xu, 2004) do not specify what quality attribute of Web Services they are assessing.
- Different researchers may use the same testing technique but name this technique differently; an example of this: Zhang (Zhang, et al. 2004a) mentioned the use of Interface Propagation Analysis (IPA) to test Web Services, while (Offutt & Xu, 2004) mentioned the use of data perturbation; both of the authors mean the same testing technique.
- Different researchers may be assessing the same quality attribute but they describe this quality attribute differently; an example of this: (Tsai, et al. 2005a), and some other researchers, mentioned they are assessing the trustworthiness of Web Services, while (Canfora, 2005) stated that the aim was to provide Service Requesters with means to build confidence that a service delivers the desired function with the expected QoS. This is similar to the trustworthiness definition but without specifying trustworthiness explicitly. (Tsai, et al. 2003) mentioned Web Service assurance which is again another related term to trustworthiness.
- Some researchers like Zhang (Zhang, et al. 2004a) state that they want increase trustworthiness of Web Services but without specifying which specific requirement of trustworthiness they are targeting.
- Some researches like (Tsai, et al. 2005a) specify that they do negative testing but they do not specify how the negative or faulty test data was generated, in other words which testing techniques have been applied.

- Very few of the Web Services testing capabilities proposed by (Bloomberg, 2002) have already been performed.

Table 4.1. Literature Survey on Fault-based Testing of Web Services

Research	Testing Techniques	Quality Attributes
(Fu, et al. 2004)	Fault injection with white box manner	Exception handling (related to robustness)
(Zhang, et al. 2004a) and (Zhang, 2004b)	Mobile agent based IPA and assertion technique to find if a Web Service meets the Service Requester requirements (specification based testing).	Trustworthy Web Service selection (does not specify which trustworthiness requirement), however, testing techniques used imply that correctness and robustness are the targeted quality attributes
(Offutt & Xu, 2004)	Boundary value testing, data perturbation, mutation testing on data rather than source code, SQL injection, using SOAP messages to supply test cases	Unspecified, however, given the testing techniques used it can be concluded that robustness and security are the targeted quality attributes
(Xu, et al. 2005)	XML Schema perturbation	Unspecified
(Zhang & Zhang, 2005b)	Boundary value testing together with faulty data perturbed from boundary value, and using WSDL for test case generation	Reliability (correctness and fault tolerance to faulty input data)
(Siblini & Mansour, 2005)	WSDL-based testing and Mutation testing	Unspecified
(Yu, et al. 2006)	Security testing for Web Services using vulnerability fault model	Security
(Looker, et al. 2007)	Fault injection with white box manner	Dependability (availability, accessibility, integrity, performance, reliability, regulatory, and security)

Table 4.2. Literature Survey on Web Services testing

Research	Testing Techniques	Quality Attributes
(Tsai, et al. 2003)	Check-in and Check-out performed by UDDI, storing test scripts in UDDI and Web Services.	Web Service assurance
(Tsai, et al. 2005a)	Unit testing, positive and negative test. Web Service composition testing, model checking, Completeness and Consistency (C&C) analysis, test case generation based on specification (OWL-S) collaborative testing, and group testing.	Trustworthiness (functionality and robustness)
(Canfora, 2005)	Regression testing	Providing Service Requesters with means to build confidence that a service delivers the desired function with the expected QoS.
(Bai & Dong, 2005)	WSDL-based testing, Random testing and boundary value testing	Unspecified

For the fault based testing technique all the research that has been surveyed in the literature is included, while for the functional testing only a few of the research is included because it is not so relevant to the research line in the thesis.

Tsai (Tsai, et al. 2005a) stated that current Web Services testing techniques assume that Web Services components have been tested properly by the Service Provider and thus focus on integration testing of composing Web Service. He also mentioned that this assumption is not acceptable if the composed Web Service need to be trustworthy because in trustworthy system every component must be verified before being used in a composite Web Services.

Tsai (Tsai, et al. 2005a) and Bai and Dong (Bai & Dong, 2005) stated that current Web Services testing techniques focus on model checking. As discussed in Chapter 3, model checking is similar to specification based testing which a kind of the validation testing.

Besides the research in the Table 4.1 and Table 4.2, some researchers address other aspects of Web Services testing such as what information should be added to the WSDL file in order to help black box and regression testing of Web Services (Tsai, et al. 2002). Specifically this research suggested adding the following to WSDL: input-output dependency, invocation sequences, hierarchal functional description, and concurrent sequence specification.

There are a number of tools to automate the Web Services testing process. Table 4.3 introduces a survey of some of these commercial tools and describes what testing techniques they use to assess which quality attributes. Most the Web Services testing tools focus on the load testing where the tool try to simulate many users using a Web Service at the same time to check if a Web Service performs as expected under this stress.

Table 4.3. Web Services Testing Tools

Vendor	Test Tool	Testing Techniques	Quality Attributes
Parasoft (Parasoft, 2007)	SOAtest	Black box, white box, unit testing, load testing, and regression testing,	Functionality (by unit, black box, and white box testing), and performance (by regression testing and load test)
Empirix (Empirix, 2007)	e- Test	Black box, load testing	Functionality, scalability, and performance
Mercury	Service Test	Black box, load test	Functionality, interoperability, and performance
Red Gate	ANTS Load	Load test	performance

4.4 Summary

Web Services Robustness testing in this thesis refers to assessing the ability of a Web Service to handle invalid input by the Service Requester.

Research in the field of Web Services testing has focused on testing the integration of composing Web Service. These are mainly based on model checking and specification based testing techniques to make sure that a Web Service does what expected. Very little research has been done on the fault based testing of Web Services that aims to detect vulnerabilities or faults and assess the robustness, security, and fault tolerance to invalid input quality attributes.

It has been noticed that Web Services testing researches may use the same testing technique but call this testing differently, such as mutation and perturbation being used

to mean the same thing. Also it is noticed that many researchers do not specify what quality attribute they are targeting or talk about the same quality attribute but in different terms.

Besides the research on Web Services testing, there exist some tools that can help to automatic the process of testing. These tools mainly focus on load testing and assessing the performance quality attribute.

Chapter 5

An Approach to WSDL-based Robustness Assessment of Web Services

5.1 Introduction

This chapter describes an approach for assessing the robustness quality attributes of Web Services. The approach depends on applying the fault-based testing techniques discussed in Chapter 3 on the Web Services in order to assess these quality attributes. The fault-based testing techniques and the input parameters specification inside WSDL are used to design test case generation rules that can facilitate systematic Web Services Quality of Service (QoS) assessment.

This thesis is concerned with the robustness faults that have the following properties:

a. Caused by the inability of the Web Service implementation to handle some test data by raising the proper exception.
b. Caused by the inability of the Web service platform to handle invalid or faulty input.

There may be other faults that may affect the robustness of Web services that are related to one of the following:

a. Faults that are related to other components of WSDL apart from the XML Schema datatype of the input parameters.
b. Faults that are related to other standards in Web services such as SOAP messages and registry.

But these faults are out of scope of this research and will be considered in future research.

5.2 Overall Architecture

This section will describe an overall architecture of the proposed approach for assessing the robustness quality attributes assessment of Web Services (See Figure 5.1).

The components of the architecture in Figure 5.1 are:

- **WSDL** is the contract or the specification of the Web service under test.
- **WS Test Case Generator** is the component that is responsible for generating test cases based on the WSDL document of the Web Service under test and the test case generation rules.

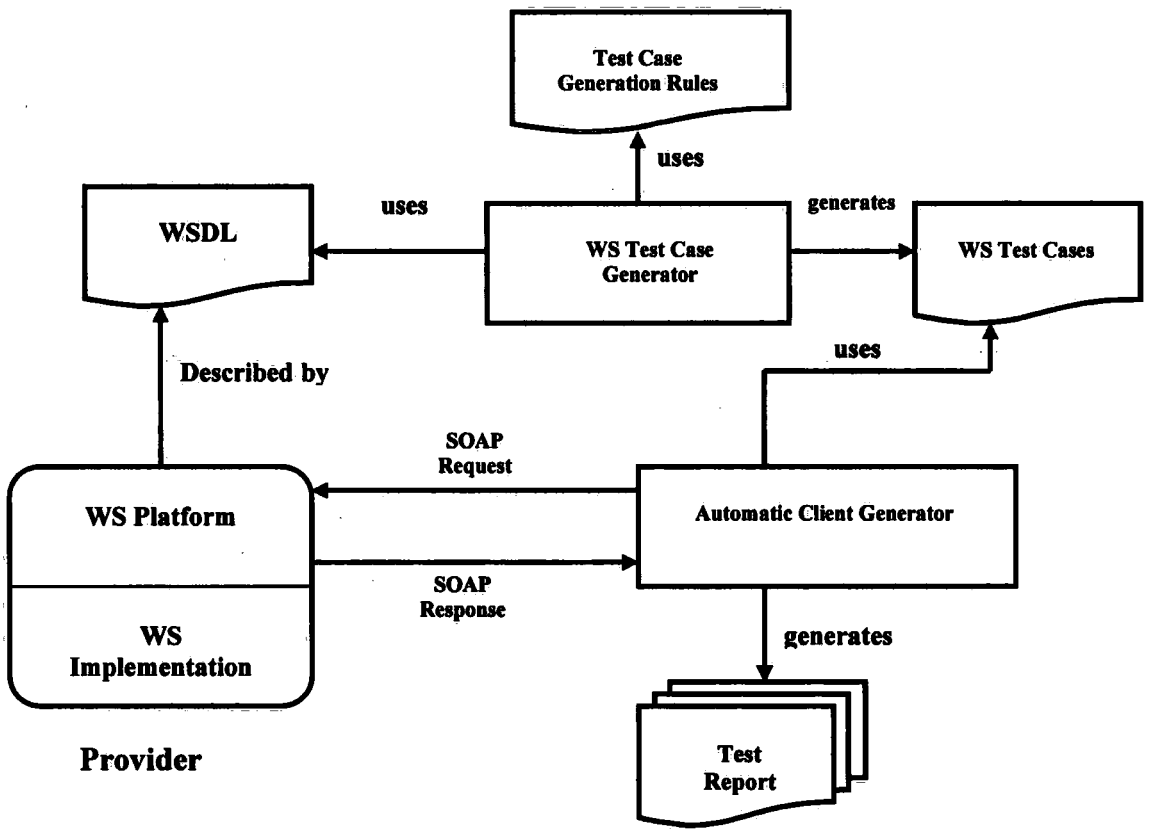


Figure 5.1 Overall architecture of the Web services robustness testing framework

- **WS Platform** is the platform or middleware that the Web Service Provider is using for his Web Service implementation. Examples of Web Service platforms are Axis (Apache Software Foundation, 2005) and GLUE (WebMethods, 2007). Some faults (or resultant failures) may be related to the platform that was used to implement a client to a Web service rather than to the Web Service implementation. For example we might send a SOAP message to a Web Service under test but the platform used does not deliver this message to the targeted Web Service due to a failure in the middleware. The SOAP message might be delivered correctly but the platform where the Web Service implementation deployed may not deliver the request to the Web Service implementation.
- **Test Case Generation Rules** are the rules that are proposed for test case generation. These rules depends on the following:
    1. Analyzing the kind of faults may affect the robustness quality attribute of Web services and that can be detected using the data inside WSDL.
    2. Analyzing what are the testing techniques that can be used to detect those faults.
    3. Analyzing the parts inside the WSDL's XML Schema-based datatypes description that can be used in testing the robustness of a Web service.
- **Test Report** is an XML document that describes the test data together with the actual response of the Web Service under test for each of the test data in each test case.
- **WS implementation** is the source code of the Web Service that is written by the Web Service Provider.

- **Automatic Client Generator** is the component that is responsible of building a client to the Web Service under test and invoking it using the test cases provided by the Web Service test case generator component. It receives the test case that was generated by the WS test case generator component and then use the information inside this test case document to send SOAP messages (over HTTP), using a certain platform or middleware, to the Web Service, and then analyze the SOAP responses and generate test results accordingly.
- **WS Test Cases** is an XML document that includes the test cases for each operation inside the WSDL of the Web Service based on the test case generation rules.

Interaction between the components in Fig. 5.1 is described in the following:

1. Test case generation rules are designed based on: input parameters' XML Schema-based datatype specification, robustness faults that may affect Web Services, the traditional testing techniques, and the quality attribute(s) being assessed.
2. Web Service Provider deploys his/her Web Service implementation in a Web Service platform.
3. The WS test case generator component uses the test cases generation rules in 1 and the WSDL document of the Web Service in 2 to generate the Web Service test case.
4. The automatic client generator will generate a client to invoke the Web Service deployed in 2 using the test case developed in 3 and then generate the test results document accordingly.

5.3 A Model for Robustness Testing of Web Services

The previous section introduced an overall architecture of the Web Services robustness testing framework. This section will give a detailed specification of the components that participate in Web Services robustness assessment and how they are related to each other (See Figure 5.2). Some of the components previously defined will be explained in more details here.

- **Operation** is the operation element inside WSDL (see chapter 2) of the specific operation under test.
- **Input and Output Message** is the input and output messages of the WSDL operation under test (see chapter 2).
- **Input and Output Parameter** are the parameters of the input and output message of an operation inside WSDL. The input parameters are specified in the part element which is a sub-element of the message element.

  List 5.1 gives an example of input parameters to an input message of an operation of WSDL. The input message called toFahrenheitRequest and this message accepts one parameter called pCentigrade of type xsd:double. (xsd: XML Schema Datatype)

```
<wsdl:message name="toFahrenheitRequest">
    <wsdl:part name="pCentigrade" type="xsd:double"/>
</wsdl:message>
```

List 5.1: An example of a simple input parameter specification inside WSDL

- **XML Schema Datatype** is the datatype specification of the input parameter to the WSDL operations. The datatype of a parameter which is represented by the

type attribute of the part element could be one of the XML Schema datatypes discussed in Chapter 2. To assure the interoperability between the Service Provider and Consumer, they both must used XML Schema to describe their data.

- **Network Protocol Stack** is the set of protocols used for communication. Network protocol stack contains the following layers: physical, link, network, transport, and application layer.
- **Quality Attribute**: The quality attributes vary between Web Services applications according to the domain where the Web Services are used and the Service Requester preference. However, this thesis is only concerned with robustness and the related attributes that include security, and fault tolerance.
- **Robustness**: The robustness of Web Services is the quality aspect of whether a Web Service continues to perform despite violating the constraints in its input parameters specification.

The other quality attributes that are related to robustness are:

a. The fault tolerance to invalid input: This means the ability of a Web service to tolerate the faults that are related to receiving an invalid input.
b. Malicious input vulnerability: this is on aspect of security quality attribute which measure if a Web service is vulnerable to an input that attempts to intrude or attack this services such as SQL injection (Offutt & XU, 2004).

So the quality attributes that also affected by the robustness are fault tolerance and security.

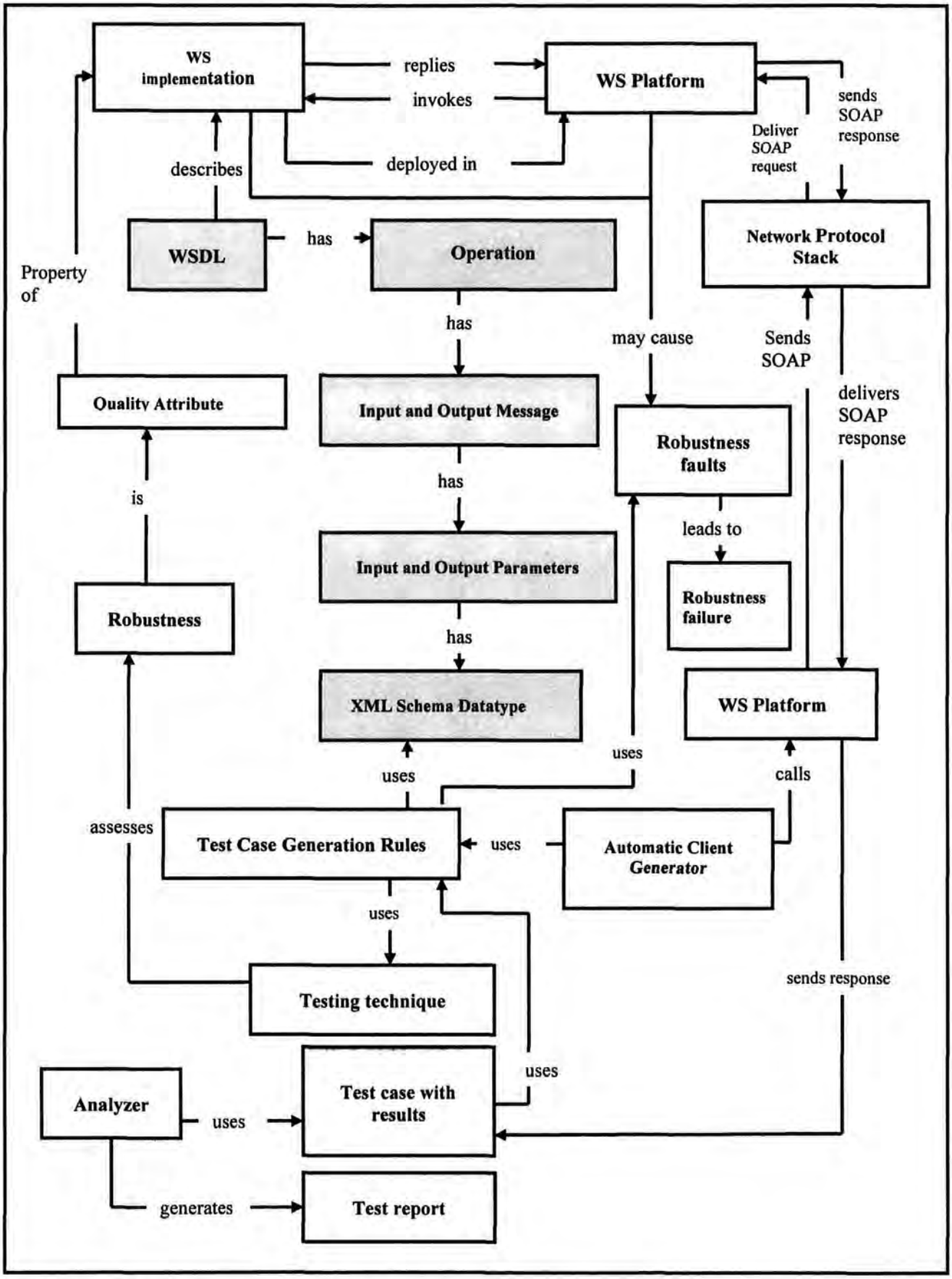


Fig. 5.2: A Model of WSDL-based Robustness Testing of Web Services

Robustness is sub-attribute or characteristic of reliability. Reliability itself is a sub-attribute of dependability and trustworthiness, so in order to assess how dependable and trustworthy a Web Service is, all sub-attribute of dependability and trustworthiness must be assessed.

- **Robustness Fault** is the fault the affects the robustness quality attribute of a Web Service.
- **Robustness Failure** is when the SOAP response is one of those as described in Fig. 5.3.

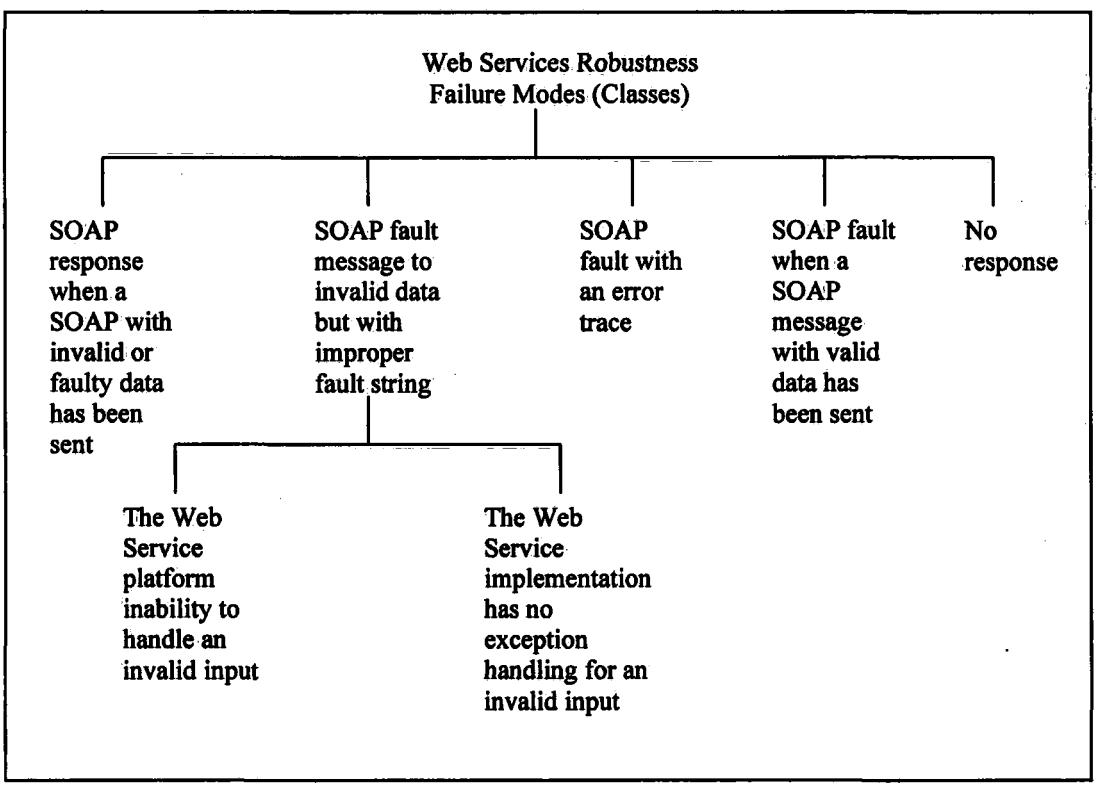


Fig. 5.3. Web Services Robustness Failure Modes

- **Test Case Generation Rules:** Test case generation rules are the rules that will be used for the test case generation for Web services. Section 5.4 will describe in details the process of test case generation for Web Services. The test cases are

described in atomic rules in order to make the process of test case generation more systematic and to enable more than one entity in the Web Service architecture to add test cases (such as the Service Provider, the Service Requester, and the Service Registry). The approach of describing test cases using atomic rules is described in (Murnane, Hall & Reed, 2005) but those rules for traditional black box testing technique. This research modified these rules by adding the information the can be extracted from WSDL to be used in testing. Also added fields that can explain the relationship between each fault, WSDL component, quality attribute, and testing technique.

For each XML Schema component that is associated with an input parameter datatype, a different testing technique will be chosen to generate test data, where testing techniques selection will depend on the characteristics of the associated component to the datatype, the following two examples will explain the idea more:

Example 1: For the *minInclusive* and *maxInclusive* constraining facets that specifies the boundaries to the numeric datatypes (such as integer datatype) boundary value based robustness testing (Jorgensen, 2002) will be used to generate test data since this testing technique deals with the boundaries of the parameters to a method.

Example 2: For the pattern constraining facet which a regular expression that constrains the characters or literals of a parameter to those that matches a specific pattern, syntax testing (Beizer, 1990) (also called input validation testing) will be used to generate test data because syntax testing is used to validate input-data which can be expressed in regular expressions or other formal forms.

Section 5.4 will explain in detail how test cases are generated to assess Web Services robustness by modifying the traditional testing techniques that can be used to violate WSDL specification.

- **Automatic Client Generator:** Client generator is the component that is responsible of building a client to the Web service under test and invoking the Web service under test using the test data.
- **Analyzer:** The analyzer is the component that compares the response of the Web service with the expected response that can be taken from the test case.
- **Test Report:** Test report is the result of the test.

Now, that all the components in Figure 5.1 have been defined, the relationships between those components can be listed:

- Robustness is considered a quality attribute to be assessed
- Quality attributes are properties of Web services under test that are deployed in a specific Web Service platform.
- Web services are described by using WSDL
- Each operation has an input message
- The WSDL component that is important in testing the robustness attribute is the XML Schema datatype of the input message parameters.
- To assess the robustness quality attribute using WSDL, the faults that affect robustness and that can be introduced by WSDL must be analyzed.
- The faults that are considered in this model are those that can be introduced a Web Service by the input parameters datatypes and their constraints.

- Test case generation rules uses the specifications of these datatypes and uses the robustness testing techniques to generate test cases.
- The client generator component will automatically use the test case generation rules to send SOAP messages to the Web services under test using a Web Service platform or SOAP implementation.
- The Web Service will reply to this message by sending a response message or a fault message using the Web Service platform that its implementation is deployed in.
- The client middleware or platform uses the network to send SOAP messages to the Web service under test, and also the client middleware receives the SOAP message, that were sent, using the network.
- The analyzer component will compare the actual response that is the expected response of each test case.
- The analyzer will then generate a test report depending on the comparison between the expected and actual response of the Web service under test.

5.4 Test Case Generation Rules

Test case generation in this thesis depends on the input parameters XML Schema datatype specification; this section will explain how these datatypes and their constraints can be used to generate test cases.

XML Schema datatypes can be categorized as:

- Built-in primitive (or derived from built-in primitive) simple datatypes
- User-defined simple datatypes.

- Complex datatypes.

Test cases will be generated depending on which of these categories an input parameter belongs.

Table 5.1 contains a schema for the test case generation rules that is proposed by this thesis. A brief description of the attributes or components of the schema in Table 5.1 is as follows:

1. **ID** attribute is a unique identifier for different rules
2. **WSDL component(s) test data is based on**; since the test cases depending on the information inside WSDL, this attribute specifies the WSDL component that the current test case is based on.
3. **Fault** attribute is the fault that the current test case assumes to detect.
4. **Traditional testing technique** describes the fault-based testing technique (See Chapter 3) that is used to generate test data to assess the fault. The fault-based testing techniques that this research uses to assess Web Services robustness include: robustness testing, syntax testing (input validation testing), equivalence partitioning, and Interface Propagation Analysis (IPA).
5. **Traditional test data generation rule** describes how the test data is generated depending on the testing technique used.
6. **Valid/Invalid** attribute used to specify if the test data are valid or invalid.
7. **WS Datatype** attribute describes the XML Schema-based datatype of the input parameter of the Web Service operation under test.
8. **WS Test Datatype** defines the datatype of the test data used in this test case. The datatype of the test data in not always the same as the Web Service datatype because for example some test cases use integer input for an operation that

accepts a string as input in order to test if the operation will produce a proper or graceful exception or not.

9. **WS test data** is the actual data that is used to in the current test case.
10. **Expected output** specifies the expected SOAP response or SOAP fault of the Web Service under test based on the current test case.
11. **Quality attribute(s) assessed** specifies the quality attribute targeted by the current test case. This research mainly concerned with the robustness quality attribute, however, other quality attribute, such as security, may also be tested by the same test case.

Table 5.1 Schema for the Test Case Generation Rules

Attribute	type	Description
1. **ID**	String	Identifier or reference of the rule
2. **WSDL Component(s) test data is based on**	enum	The WSDL component(s) this test data is based on which could be the input parameter datatype or the constraining facets for the input parameter datatype
3. **Fault**	enum	The fault that the test data suppose to detect
4. **Traditional Testing Technique**	enum	The traditional testing technique used in the rule, Testing_Technique::= EP | RT | IPA | ST | SI Where EP = Equivalent Partitioning (Myer, 1979) RT = Robustness Testing (Jorgensen, 2002) IPA = Interface Propagation Analysis (Voas & McGraw, 1998a) (Voas, 1998b) ST = Syntax Testing (Beizer, 1990)
5. **Traditional test data generation rule**	String	Description of how the test data is generated using the used traditional testing technique
6. **Valid/Invalid**	enum	whether the test data chosen valid or not
7. **WS Datatype**	datatype	Defines the Web service datatype of the input parameter tested.
8. **WS Test Datatype**	datatype	Defines the Web service datatype of the test data which might be the same as the Web service datatype or different.
9. **WS test data**	Depends on WS Test Datatype	Defines the actual data used for testing
10. **Ex pected output**	String	Defines what is the expected response SOAP message of the Web service under test
11. **Quality attribute (s) assessed**	enum	Defines the quality attribute this test data aims to assess which could be robustness and/or security and/or fault tolerance.

5.5 Generating Test Cases for Primitive (or derived from primitive) Simple Datatypes

W3C XML Schema primitive (or derived from primitive) simple datatypes (see Figure 2.5) can be categorized as String datatypes, Numeric datatypes, Date-Time datatypes, and Boolean. Table 5.2 describes the datatypes included in each of these categories. A description of each of these datatypes, together with the value space of each of them, can be found in (W3C, 2004c) and (Vlist, 2002).

Designing test data for primitive or derived from primitive simple datatype is more difficult than designing test data for user-derived and complex datatypes because there are no constraining facets and other schema components that can help in designing the test cases.

To generate test data for robustness assessment when the input message parameter to a Web Service of simple datatype (primitive or derived from primitive) will depend on changing the datatype of the input parameter, supplying a null or empty parameter, or using the upper and lower limits of values. Each datatype category in Table 5.2 will be considered in turn and these changes will be applied to them.

Table 5.2. W3C XML Schema Primitive or Derived from Primitive Simple Datatypes

Numeric Datatypes	String Datatypes	Date-Time Datatypes	Boolean
decimal	string	duration	boolean
integer	normalizedString	dateTime	
int	token	time	
byte	language	date	
short	Name	gMonthDay	
long	NMTOKEN	gYearMonth	
nonPositiveInteger	NCName	gYear	
nonNegativeInteger	ID	gMonth	
unsignedInt	IDREF	gDay	
unsignedByte	Entity		
unsignedShort	base64Binary		
unsignedLong	hexBinary		
positiveInteger	anyURI		
negativeInteger	QName		
float	NOTATION		
double			

5.5.1 Test Cases Generation Schema

This section will describe the tables of the test case generation schema that was used to generate test cases for primitive or derived from primitive datatype (See Table 5.3).

To explain why the specific test cases in Table 5.3 have been used with the primitive or derived from primitive datatypes, a formal description of test case rules selection will be given:

For the primitive datatypes in Table 5.2, let:

N represents *Numeric* Datatypes

S represents *String* Datatypes

DT represents *Date-Time* Datatypes

B represents *boolean*

The test case generation rules in Table 5.3 are produced as follows:

{N, DT, B} replace with S, produce → {*String_Replacement*}

{S, DT, B} replace with N, produce → {*Numeric_Replacement*}

{N, S, B} replace with DT, produce → {*Date_Time_Replacement*}

{N, S, DT} replace with B, produce → {*Boolean_Replacement*}

{S, DT, B} replace with N, produce → {*Numeric_Replacement*}

{N, S, DT, B} replace with null, produce → {null_Replacement}

{N, DT} replace with boundary values, produce → {Max_Value, Above_Max, Less_Max, Min_Value, Above_Min, Less_Min}

{N} replace with Zero, produce → {Zero_Input}

{N} replace with NaN, produce → {NaN_Replacement}

{S} replace with extreme values, produce → {Large_String, Empty_String}

Table 5.3 (a): Test Case Generation Rules for Primitive or Derived from Primitive Simple Datatypes

ID	String_Replac-ement	Numeric_Repl-acement	Date-Time_Replace ment	Boolean_Rep-lacement
WSDL Component(s) test data is based on	Operation input message parameter's datatype	Operation input message parameter's datatype	Operation input message parameter's datatype	Operation input message parameter's datatype
Fault	Lack of validation of input datatype	Lack of validation of input datatype	Lack of validation of input datatype	Lack of validation of input datatype
Traditional Testing Technique	EP & IPA	EP & IPA	EP & IPA	EP & IPA
Traditional test data generation rule	Replace the input parameter with String	Replace the input parameter with Numeric	Replace the input parameter with Date-Time	Replace the input parameter with Boolean
Valid/Invalid	Invalid	Invalid	Invalid	Invalid
WS Datatype	Numeric, Date-Time, and Boolean	String, Date-Time, and Boolean	String, Numeric, and Boolean	Numeric, String, and Date-Time
WS Test Datatype	String	Numeric	Date-Time	Boolean
WS test data	Random String	Random Numeric	Random Date-Time	Random Boolean
Expected output	Fault message with proper fault string	Fault message with proper fault string	Fault message with proper fault string	Fault message with proper fault string
Quality attribute(s) assessed	1. Platform Robustness (handling invalid input) 2. Platform Security (input manipulation vulnerability) 3. Platform Fault tolerance to wrong input	1. Platform Robustness (handling invalid input) 2. Platform Security (input manipulation vulnerability) 3. Platform Fault tolerance to wrong input	1. Platform Robustness (handling invalid input) 2. Platform Security (input manipulation vulnerability) 3. Platform Fault tolerance to wrong input	1. Platform Robustness (handling invalid input) 2. Platform Security (input manipulation vulnerability) 3. Platform Fault tolerance to wrong input

Table 5.3 (b)

ID	null_Input	Max_Value	Above_Max	Less_Max
WSDL Component test data is based on	Operation input message parameter's datatype	Operation input message parameter's datatype	Operation input message parameter's datatype	Operation input message parameter's datatype
Fault	Lack of validation of null input	Lack of ability to handle large numbers and boundary fault	Lack of ability to handle large numbers and boundary fault	Lack of ability to handle large numbers and boundary fault
Traditional Testing Technique	EP & IPA	RT	RT	RT
Traditional test data generation rule	Replace the input parameter with null	Replace the input parameter with maximum allowed number	Replace the input parameter with maximum allowed number + 1	Replace the input parameter with maximum allowed number - 1
Valid/Invalid	Invalid	Valid	Invalid	Valid
WS Datatype	All	Numeric, Date-Time	Numeric, Date-Time	Numeric, Date-Time
WS Test Datatype	null	Same as WS Datatype	Numeric	Same as WS Datatype
WS test data	null	Maximum allowed number of the WS Datatype	Maximum allowed number of the WS Datatype + 1	Maximum allowed number of the WS Datatype - 1
Expected output	Fault message with proper fault string	Response message	Fault message with proper fault string	Response message
Quality attribute(s) assessed	1.1. Platform Robustness (handling invalid input) 2. Platform Security (input manipulation vulnerability) 3. Platform Fault tolerance to wrong input	1. WS implementation robustness (handling stressful environmental condition) 2. WS implementation security	1. Platform Robustness (handling invalid input) 2. Platform Security (input manipulation vulnerability) 3. Platform Fault tolerance to wrong input	1. WS implementation robustness (handling stressful environmental condition) 2. WS implementation security

Table 5.3 (c)

ID	Min_Value	Less_Min	Above_Min	Zero_input
WSDL Component test data is based on	Operation input message parameter's datatype	Operation input message parameter's datatype	Operation input message parameter's datatype	Operation input message parameter's datatype
Fault	Boundary fault	Boundary fault	Boundary fault	Zero input fault
Traditional Testing Technique	RT	RT	RT	EP
Traditional test data generation rule	Replace the input parameter with minimum allowed number	Replace the input parameter with minimum allowed number - 1	Replace the input parameter with minimum allowed number + 1	Replace the input parameter with zero
Valid/Invalid	Valid	Invalid	Valid	Valid
WS Datatype	Numeric, Date-Time	Numeric, Date-Time	Numeric, Date-Time	Numeric where zero is valid
WS Test Datatype	Same as WS Datatype	Numeric	Same as WS Datatype	Numeric
WS test data	Minimum allowed number of the WS Datatype	Minimum allowed number of the WS Datatype - 1	Minimum allowed number of the WS Datatype + 1	zero
Expected output	Response message	Fault message with proper fault string	Response message	Response message
Quality attribute(s) assessed	1. WS implementation robustness (handling stressful environmental condition) 2. WS implementation security	1. Platform Robustness (handling invalid input) 2. Platform Security (input manipulation vulnerability) 3. Platform Fault tolerance to wrong input	1. WS implementation robustness 2. WS implementation security	1. WS implementation robustness 2. WS implementation security

Table 5.3 (d)

ID	NaN_Replacement	Large_String	Empty_String
WSDL Component test data is based on	Operation input message parameter's datatype	Operation input message parameter's datatype	Operation input message parameter's datatype
Fault	Lack of validation of NaN value	buffer overflow	Lack of validation of empty String
Traditional Testing Technique	EP	EP	EP
Traditional test data generation rule	Replace the input parameter with NaN	Replace the input parameter with big String	Replace the input parameter with empty String
Valid/Invalid	Valid	Valid	Valid
WS Datatype	Numeric {float, double}	String	String
WS Test Datatype	Same as WS Datatype	Same as WS Datatype	String
WS test data	NaN	A random big String	Empty String
Expected output	Response message	Response message	SOAP fault message with proper exception handling message in the fault string
Quality attribute(s) assessed	1. WS implementation robustness 2. WS implementation security	1. WS implementation robustness 2. WS implementation security	1. WS implementation robustness 2. WS implementation security

Table 5.4. Test Cases with Valid Data for Primitive or Derived from Primitive Datatypes

ID	Valid_Data
WSDL Component test data is based on	Operation input message parameter's datatype
Fault	Lack of ability to handle valid input
Traditional Testing Technique	Validation testing
Traditional test data generation rule	Provide a valid input
Valid/Invalid	Valid
WS Datatype	All
WS Test Datatype	Same as WS Datatype
WS test data	Random value of the WS Datatype
Expected output	Response Message
Quality attribute(s) assessed	1. WS implementation robustness 2. WS implementation security 3. WS functionality

5.5.2 Example of Test Case Generation

To give a detailed description of how test cases are generated, the first test case (column) in Table 5.3 will be discussed. In Table 5.3, the first test case is *String_Replacement* and it has been designed as follows:

1. The ID of this test case is *String_Replacement*.
2. Since the input parameter datatype is primitive (or derived from primitive) then there are no constraining facets for this parameter, then the WSDL component this test case is based on is only the datatype of the input parameter.
3. The fault that this test case is to detect is the lack of validation of input datatype. This means that this test case assesses if the Web Service operation under test is robust when the input datatype is not the same as expected by the Web Service (as described in WSDL).
4. Since this test case is to detect the lack of validation of input datatype and it must change the input parameter datatype, then the testing technique that is used to perturb the input data is IPA. Changing the datatype is considered an invalid equivalent class in an equivalent partitioning testing (Myers, 1979).
5. The traditional test data generation rule in IPA is to perturb the input parameter by changing the datatype to string.
6. This test case is invalid because it is used to send invalid input to the Web Service under test.
7. The Web Service XML Schema datatype in this test case is *Numeric, Data-Time,* and *Boolean* because the input parameter is replaced by String so the input parameter must be different than String.
8. The Web Service test datatype is String.

9. The test data is a random String that can be generated using a random string generation function.
10. The expected output in this test case is that the Web Service sends a fault message that describes to the Service Requester that the Web Service does not expect a string but rather the actual input parameter datatype as described in WSDL.
11. The quality attributes assessed are, first: Web Service platform robustness, since the datatype is different than the expected datatype in WSDL. Then the Web Service platform must be robust enough and not send the SOAP request to the Web Service implementation but rather send a fault message directly to the Service Requester. Second: security, the Web Service platform or the Web Service source code may raise an uncaught exception causing a stack trace. This stack trace might then be used by malicious Service Requesters to harm a Web Service. So this test case assess if the Web Service under test is vulnerable to such attacks by checking its response to an invalid datatype. Third: fault tolerance to wrong input, this test case assess if the Web Service under test can handle the wrong datatype fault without causing a failure to the Web Service.

5.5.3 Detailed Description of Test Case Generation

This section will explain in more details how test cases are generated in Table 5.3 by discussing the components (rows) of this table that need more explanation:

Fault:

Table 5.3 shows how different faults can be detected when the input parameter to a Web service is of primitive or derived from primitive datatype.

The rules in this table are concerned with the following faults:

The rules in this table are concerned with the following faults:

- **Lack of validation to input datatype:**

  These faults occur when the input parameter to a Web service is of a datatype that is different than the expected datatype. For example, an input message for a certain operation expects an integer parameter while the input was of type string. If the Web Service platform contains a validation to the input datatype and sends a proper fault message when such faults occur, without sending the request to the Web Service implementation, then no robustness failure will result. Otherwise it is the responsibility of the Web Service implementation to raise an exception to this invalid datatype to prevent a robustness failure.

- **Lack of validation of null input:**

  The Web Service platform must validate a null input in order to be robust to this kind of faults.

- **Lack of validation of empty string:**

  The detection of this fault is not the responsibility of the Web Service platform because it is a valid input and the method request inside WSDL must be given to the Web Service implementation.

  The faults that we just discussed are considered as input manipulation faults or vulnerabilities. These faults occur when unexpected datatypes are used as input to a Web service.

  There are other types of input manipulation vulnerabilities or faults such as SQL injection but this thesis is not concerned with these faults because our main target to assess the robustness of a Web service using the information inside

WSDL rather than assessing the security of a Web service. Only the security vulnerabilities that are related to robustness also are discussed in this thesis.

- **Boundary fault**

Experience shows that test cases that explore boundary conditions can detect more fault that test cases that do not (Myers, 1979). For this reason some test cases have been designed to explore the boundaries of the Web Service operation's input parameter XML Schema datatype.

The testing techniques that are used to detect such faults are robustness testing (Jorgensen, 2002) and boundary-value analysis (Myers, 1979). For the test cases: *Above_Max* and *Less_Min*, the Web Service platform robustness (input vulnerability and fault tolerance wrong input) is tested because the platform should be robust and not send the operation request to the Web Service implementation. The test cases: *Max_Numeric, Min_Numeric, Above_Min,* and *Less_Max*, are used to check if the Web Service implementation has no boundary faults (or robust to such kind of faults).

To apply the boundary value based test cases, the boundaries of the Numeric XML Schema datatypes in table 5.2 must be found. Table 5.4 summarizes the boundary value for each of the Numeric datatypes that have constraints on the number of digits as specified by W3C standard for XML Schema datatypes (W3C, 2004c). The Numeric datatypes that are not mentioned in Table 5.4 have unconstrained length and so can not be tested for binary faults.

- **Lack of ability to handle zero input**

  This fault occurs if the Web Service implementation is vulnerable to zero input, or possibly if the Web Service implementation has no divide by zero exception handler.

Traditional Testing Techniques:

The traditional testing techniques are chosen depending on the fault that a test case is supposed to detect. The fault that the rules *String_Replacement, Numeric_Replacement, Date-Time_Replacement,* and *Boolean_Replcement* are to detect is the lack of validation of input datatype. After a survey on the traditional testing techniques in Chapter 3, it has been found that the testing techniques that can assess if a system has this kind of faults are equivalent partitioning and interface propagation analysis (IPA).

For the boundary faults, the testing technique that is used to assess such faults is the boundary value based robustness testing (Jorgensen, 2002).

The fault that the rule *null_Replacement* supposes to detect is lack of validation of null and the traditional testing technique that is used to detect such faults is equivalent partition.

The same analysis can be easily followed for the other test cases.

WSDL Component test data is based on:

Test case generation for simple primitive (or derived from primitive) datatype depends only on the input parameter datatype.

Expected Output:

For the test cases that change the datatype of the input parameter or send a null input, the expected output is that the Web Service platform will not send this request to the

Web Service implementation and rather send a response to the Web Service Requester with a proper fault message such as: "Wrong type, this operation expects integer datatype but it received a string".

Table 5.5. Numeric XML Schema Datatypes Boundaries

Numeric Datatype	Min Allowed Value	Max Allowed Value
nonPositiveInteger	Undefined	0
long	-9223372036854775808	9223372036854775807
nonNegativeInteger	0	Undefined
negativeInteger	Undefined	-1
int	-2147483648	2147483647
unsignedLong	0	18446744073709551615
positiveInteger	1	Undefined
short	-32768	32767
unsignedInt	0	4294967295
byte	-128	127
unsignedShort	0	65535
unsignedByte	0	255
float	1.4E-45, -INF	3.4028235E38, INF
double	4.9E-324, -INF	1.7976931348623157E308, INF

Quality attribute(s) assessed:

Some test cases assess whether the Web Service platform has the ability to handle requests with invalid data without sending the request to the Web Service implementation. In these test cases, the quality attribute assessed are:

1. The robustness of the Web Service platform: platform ability to handle invalid data.
2. Platform Security: if the Web Service platform is vulnerable to some input, then it will send a stack trace to the Service Requester that will enable malicious Requesters to harm the Web Service.
3. Platform Fault tolerance to wrong input: checking if the platform can tolerate wrong or invalid input without causing a failure.

5.6 Generating Test Cases for User-derived Datatypes

User-derived are created by restricting a built in (or derived from built in) datatype (called the base type) using constraining facets. Descriptions of all the constraining facets are found on the W3C specification (W3C, 2004c).

Constraining facets are used to restrict a base datatype by specifying some characteristics of this datatype like the maximum and minimum length allowed for a string value and the maximum and minimum allowed numbers for a numeric value. For example constraining facets may specify that a certain integer number may only assume numbers between 1 and 100, and so on. Table 2.2 gave a brief description of all the constraining facets (W3C, 2004b). Different simple datatypes have different

constraining facets, for example, string datatypes have the constraining facets: length, minLength, maxLength, pattern, enumeration, and whiteSpace.

The approach for test data generation for this kind of datatypes depends on:

- The constraining facets (W3C, 2004c).
- The base type (the datatype from which the user derived datatype was derived)

5.6.1 Test Case Generation Schema

For each constraining facet for the different datatypes an analysis has been carried out on what faults may be caused by violating the datatype's constraining facets and also what test cases should be used to detect these faults.

The base datatypes of user-derived datatypes (that are primitive or derived from primitive) will have the same categories in Table 5.2, however, Numeric datatypes category will be divided into *Decimal* and *Float* categories, where each contains the following datatypes:

Decimal: *decimal, integer, nonPositiveInteger, long, nonNegativeInteger, nagativeOnteger, int, unsignedLong, positiveInteger, short, unsignedInt, byte, unsignedShort, and unsignedByte.*

Float: float and double.

This categorization is done because the datatypes in the Decimal category have different constraining facets than the datatypes in the Float category and the test case generation will depend on these constraining facets.

So, to generate test cases for user-derived datatypes there will be the following categories for the base datatypes: Decimal, Float, String, date-Time, and Boolean.

5.6.1.1 Test Cases based on the Numeric Boundaries Constraining Facets

The Numeric Boundaries constraining facets include: minInclusive, minExclusive, maxInclusive and maxExclusive. The minInclusive and minExclusive constraint specifies an inclusive and exclusive lower bounds for the value space of a datatype while maxInclusive and maxExclusive specifies an inclusive and exclusive upper bounds for the value space of a datatype. Table 5.6 describes how test data are generated based on these constraining facets.

Since the numeric boundaries facets are related to the lower and upper bounds of an input parameter to a Web Service then it was natural to use boundary value based robustness testing technique (Jorgensen, 2002) to generate test data.

Table 5.6 (a): Test Case Generation for User-derived datatype Numeric Boundaries

ID	Min_Value	Above_Min	Less_Min	Min_Value
WSDL Component test data is based on	minInclusive	minInclusive	minInclusive	minExclusive
Fault	Boundary fault	Boundary fault	Boundary fault	Boundary fault
Traditional Testing Technique	RT	RT	RT	RT
Traditional test data generation rule	Replace the input parameter with minimum allowed number	Replace the input parameter with value just above minimum allowed number	Replace the input parameter with value just below minimum allowed number	Replace the input parameter with minimum allowed number
Valid/Invalid	Valid	Valid	Invalid	Valid
WS base Datatype	Decimal, Float	Decimal, Float	Decimal, Float	Decimal, Float
WS Test Datatype	Same as WS Datatype	Same as WS Datatype	Same as WS Datatype	Same as WS Datatype
WS test data	minInclusive value	minInclusive value + 1	minInclusive value - 1	minExclusive value + 1
Expected output	Response message	Response message	Fault message with proper fault string	Response message
Quality attribute(s) assessed	1. WS implementation robustness 2. WS implementation security	1. WS implementation robustness 2. WS implementation security	1. WS implementation robustness 2. WS implementation security	1. WS implementation robustness 2. WS implementation security

Table 5.6 (b)

ID	Above_Min	Less_Min	Max_Value	Above_Max
WSDL Component test data is based on	minExclusive	minExclusive	maxInclusive	maxInclusive
Fault	Boundary fault	Boundary fault	Boundary fault	Boundary fault
Traditional Testing Technique	RT	RT	RT	RT
Traditional test data generation rule	Replace the input parameter with value just above minimum allowed number	Replace the input parameter with value just below minimum allowed number	Replace the input parameter with maximum allowed number	Replace the input parameter with value just above maximum allowed number
Valid/Invalid	Valid	Invalid	Valid	Invalid
WS base Datatype	Decimal, Float	Decimal, Float	Decimal, Float	Decimal, Float
WS Test Datatype	Same as WS Datatype	Same as WS Datatype	Same as WS Datatype	Same as WS Datatype
WS test data	minExclusive value + 2	minExclusive value	maxInclusive value	maxInclusive value + 1
Expected output	Response message	Fault message with proper fault string	Response message	Fault message with proper fault string
Quality attribute(s) assessed	1. WS implementation robustness 2. WS implementation security	1. WS implementation robustness 2. WS implementation security	1. WS implementation robustness 2. WS implementation security	1. WS implementation robustness 2. WS implementation security

Table 5.6 (c)

ID	Less_Max	Max_Value	Above_Max	Less_Max
WSDL Component test data is based on	maxInclusive	maxExclusive	maxExclusive	maxExclusive
Fault	Boundary fault	Boundary fault	Boundary fault	Boundary fault
Traditional Testing Technique	RT	RT	RT	RT
Traditional test data generation rule	Replace the input parameter with value just below maximum allowed number	Replace the input parameter with maximum allowed number	Replace the input parameter with value just above maximum allowed number	Replace the input parameter with value just below maximum allowed number
Valid/Invalid	Valid	Valid	Invalid	Valid
WS base Datatype	Decimal, Float	Decimal, Float	Decimal, Float	Decimal, Float
WS Test Datatype	Same as WS Datatype	Same as WS Datatype	Same as WS Datatype	Same as WS Datatype
WS test data	maxInclusive value - 1	maxExclusive value - 1	maxExclusive value	maxInclusive value - 2
Expected output	Response message	Response message	Fault message with proper fault string	Response message
Quality attribute(s) assessed	1. WS implementation robustness (handling stressful environmental condition) 2. WS implementation security	1. WS implementation robustness (handling stressful environmental condition) 2. WS implementation security	1. WS implementation robustness 2. WS implementation security	1. WS implementation robustness (handling stressful environmental condition) 2. WS implementation security

5.6.1.2 Test Cases based on the String Length Constraining Facets

The String Length constraining facets include *length*, *minLength*, and *maxLength*. The length constraining facet defines a fixed length for a String datatype (See table 5.2).

The length is measured in number of characters for all the *String* datatypes except *hexBinary* and *base64Binary* where the length is measured in bytes. maxLength and

minLength specify the maximum and minimum length of a *String* also measured in character or byte like the length.

Table 5.7. Test Case Generation for User-derived datatype String Length Constraining Facets

ID	Different_Length	Longer_String1	Longer_String2	Shorter_String
WSDL Component test data is based on	length constraining facet	length constraining facet	maxLength constraining facet	MinLength constraining facet
Fault	Lack of Validating of String length	Lack of Validating of String length	Lack of Validating of String length	Lack of Validating of String length
Traditional Testing Technique	ST	ST	ST	ST
Traditional test data generation rule	NA	Add extra letter to the string input	Add extra letter to the string input	Remove one letter from the String
Valid/Invalid	Invalid	Invalid	Invalid	Invalid
WS base Datatype	String	String	String	String
WS Test Datatype	Same as Web Service DataType	Same as Web Service DataType	Same as Web Service DataType	Same as Web Service DataType
WS test data	Random string of len != length constraining facet value	Random string of len = length constraining facet value + 1	Random string of len = Maxlength constraining facet value + 1	Random string of len = minLength constraining facet - 1
Expected output	Fault message with proper fault string	Fault message with proper fault string	Fault message with proper fault string	Fault message with proper fault string
Quality attribute(s) assessed	1. WS implementation robustness 2. WS implementation security	1. WS implementation robustness 2. WS implementation security	1. WS implementation robustness 2. WS implementation security	1. WS implementation robustness 2. WS implementation security

Test case generation rules corresponding to the other constraining facets, that include *pattern, enumeration, whitespace, totalDigits, and fractionDigit*, will not be discussed in this thesis. However the rules for these constraints can easily be concluded using the same approach that was used with the other constraining facets in Section 5.6.1.1 and 5.6.1.2.

5.7 Generating Test Data for Complex Datatypes

Complex datatype consists of a group of Simple and User-derived datatypes. If the input parameter to a Web service of complex datatype then for each of the Simple and User-derived datatype of its sub-elements, the relevant test data rules are chosen as explained in section 5.1 and 5.2 and then the cross product for the test data of each of those parts are computed. The discussion of the test data generation for Web Service when the input parameter is of complex XML schema datatype will be left to a future work.

5.8 Summary

This Chapter described the approach of test case generation for Web Services that is proposed in this thesis. This approach is based on analyzing the input parameter XML Schema datatype and then finding the robustness faults that may be resulted by violating the formal specifications of this datatype. Test cases generation schema was developed depending on these faults and on the testing techniques that can be used to detect such faults. The input parameters datatype was categorized to primitive, user-derived, and complex, and test case generation rules was discussed for each primitive and user-derived datatypes only while complex datatypes will be discussed in a future work.

Chapter 6

WS-Robust: Web Service Robustness Testing Tool

6.1 Introduction

WS-Robust (Web Services Robustness) tool is an implementation of the proposed approach of test case generation to assess Web Services robustness and other related attributes proposed in Chapter 5.

The implementation is divided into three phases. The first phase is to build a database of test cases, depending on the rules in Table 5.3, Table 5.4, Table 5.6, and Table 5.7 for the simple primitive (or derived from primitive) XML Schema datatypes (shown in Figure 2.5), and user-derived XML Schema datatypes. The second phase is to write a code that can accept any WSDL document as an input and then generate the test cases. The third phase will generate clients to the Web Service under test based on the generated test cases. Fig. 6.1 represents an overall architecture of WS-Robust that describes these three phases.

6.2 Building the Rules Database

A Java program has been built that is used to manual add the rules of test case generation in order to be used by the test case generation mechanism that will be described in Section 6.3.

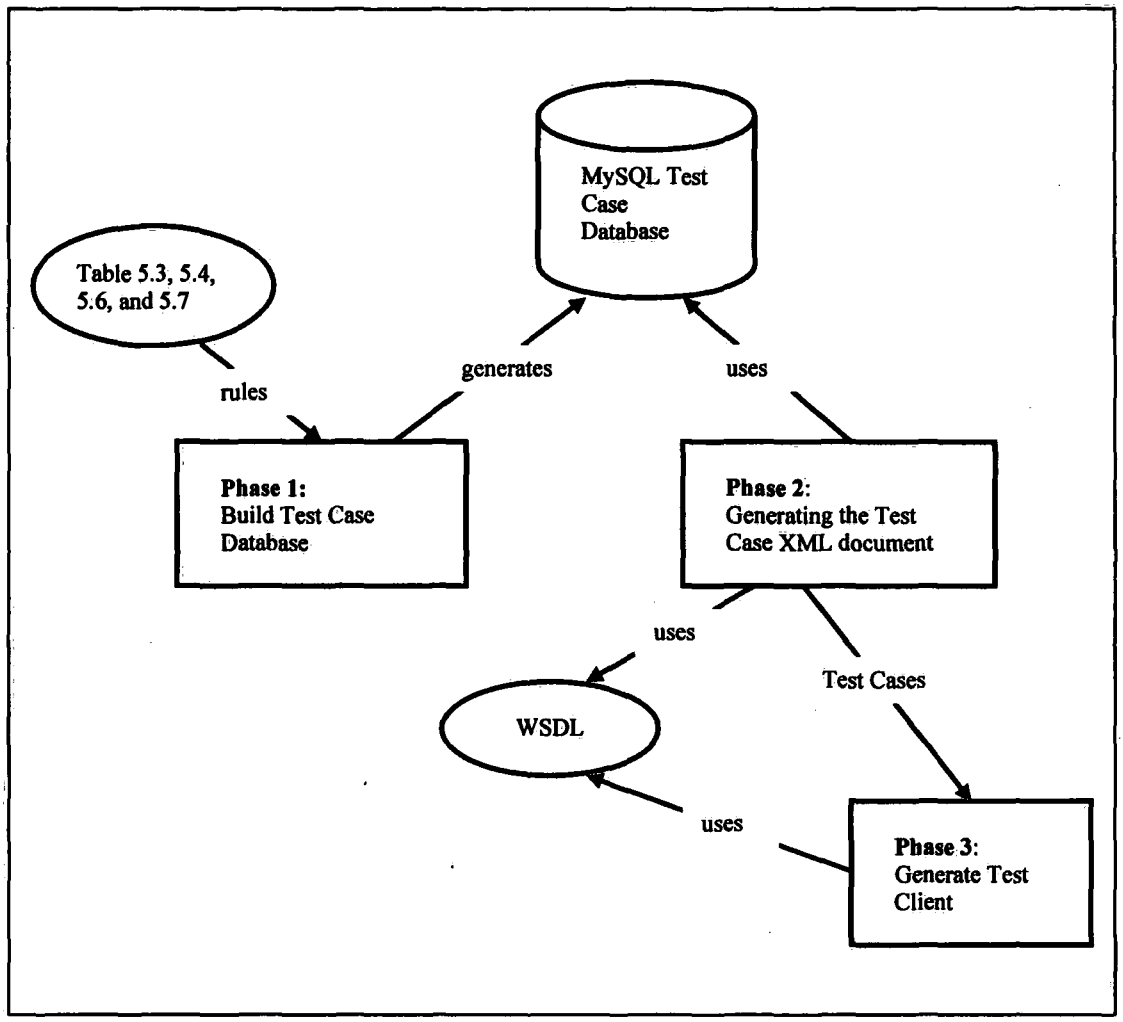


Fig. 6.1. WS-Robust Overall Architecture

A Java GUI has been implemented (see Fig. 6.2) that emulates Table 5.1. The resulted data is stored in a test cases database.

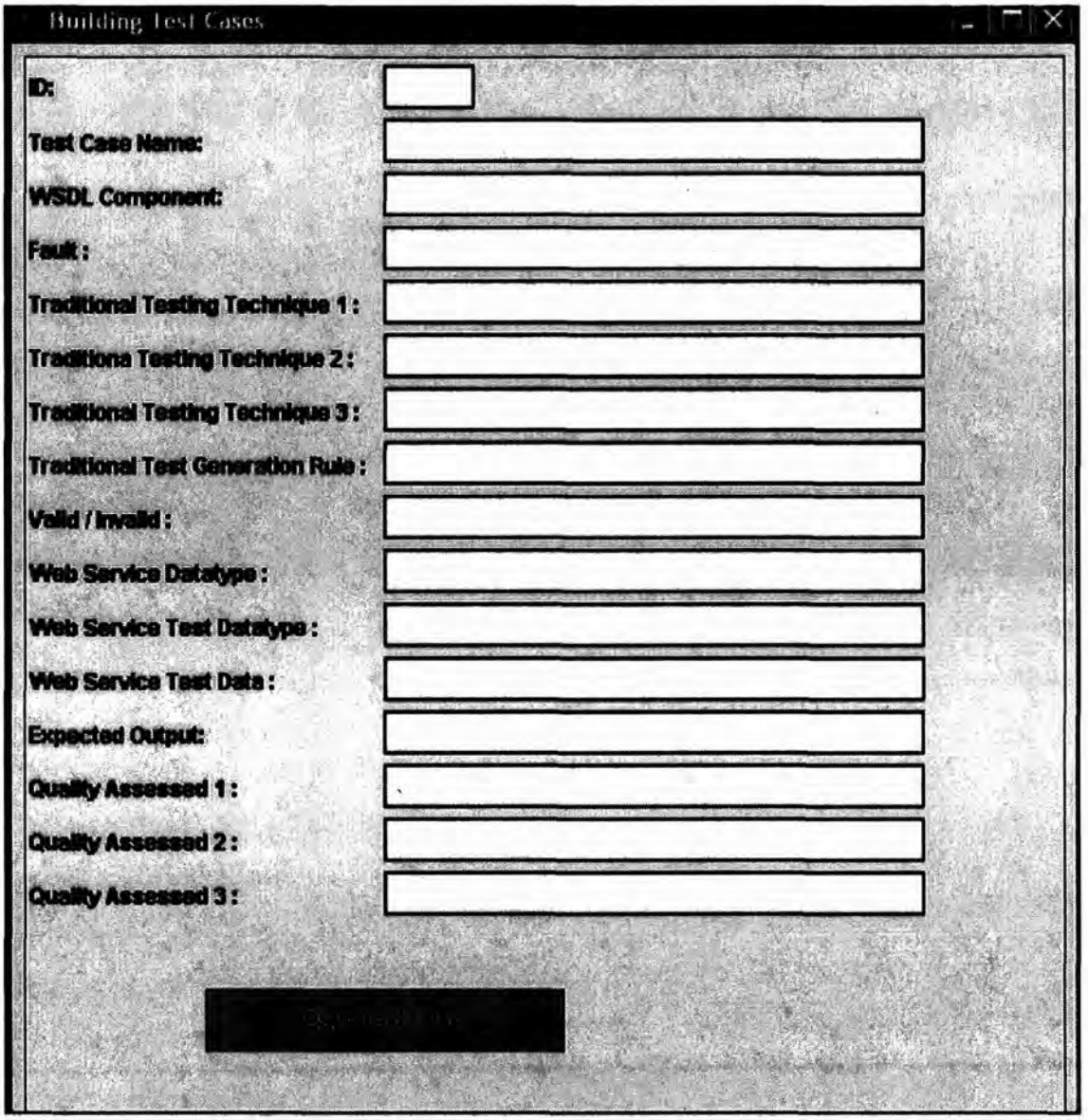


Fig. 6.2 Web Services Test Cases Building GUI

6.2.1 Configuration

The GUI is implemented using

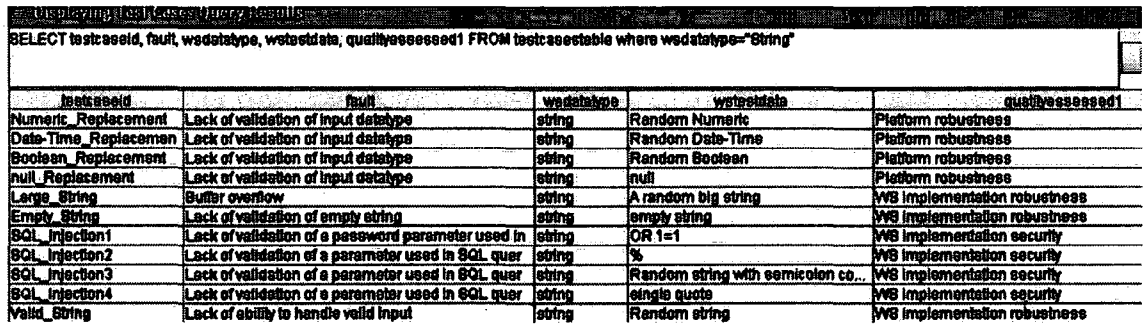
- Java 1.5.0_06.
- MySQL version 1.4.

6.2.2 Inserting the Test Cases

Using the GUI in Fig. 6.2, 434 test cases have been inserted for primitive or derived from primate simple datatypes using Table 5.3 and 5.4 of test case generation rules and 289 test cases have been inserted for user derived datatypes using Table 5.6 and Table 5.7.

6.2.3 Querying the Test Case Rules

The WS-Robust tool enables Web Service Provider or Requester to display and query the test cases rules. Fig. 6.3 shows an example of selecting the test cases rule for the Web Service with *string* input parameter.



SELECT testcaseId, fault, wsdatatype, wstestdata, qualityassessed1 FROM testcasestable where wsdatatype="String"

testcaseId	fault	wsdatatype	wstestdata	qualityassessed1
Numeric_Replacement	Lack of validation of input datatype	string	Random Numeric	Platform robustness
Date-Time_Replacemen	Lack of validation of input datatype	string	Random Date-Time	Platform robustness
Boolean_Replacement	Lack of validation of input datatype	string	Random Boolean	Platform robustness
null_Replacement	Lack of validation of input datatype	string	null	Platform robustness
Large_String	Buffer overflow	string	A random big string	WS implementation robustness
Empty_String	Lack of validation of empty string	string	empty string	WS implementation robustness
SQL_Injection1	Lack of validation of a password parameter used in	string	OR 1=1	WS implementation security
SQL_Injection2	Lack of validation of a parameter used in SQL quer	string	%	WS implementation security
SQL_Injection3	Lack of validation of a parameter used in SQL quer	string	Random string with semicolon co...	WS implementation security
SQL_Injection4	Lack of validation of a parameter used in SQL quer	string	single quote	WS implementation security
Valid_String	Lack of ability to handle valid input	string	Random string	WS implementation robustness

Fig. 6.3. Displaying and Querying the Test Rules

6.3 Test Cases Generation Mechanism

This section will describe the test cases generation mechanism for primitive datatypes and for user derived datatypes based on the test cases rules database that was created in Section 6.2.

6.3.1 Configuration

To implement the test cases generation mechanism for a specific WSDL document, the following programming language, API, plug-in, parser, and database have been used.

- Java version 1.5.0_06.
- WSDL4J (Java API for WSDL) Version 1.4.
- Eclipse plug-in that provide an API and implementation for XML Schema.
- Document Object Model (DOM) (W3C, 2005) XML parser.
- MySQL version 1.4.

6.3.2 Scenario

The scenario of test case generation is as follows:

1. The WS-Robust user (a Service Provider or a Service Requester) are prompted to enter the WSDL document location.
2. The WSDL document is parsed using DOM.
3. Create a new XML document that will be used to insert the test cases. This XML document will be called the Test Case document henceforth in this scenario.
4. Obtain the name of the Web Service using the *name* attribute of the *service* element inside WSDL (See Fig. 2.6 and List 2.6).
5. Obtain the address of the Web Service using the *address* element inside WSDL (See Fig. 2.6 and List 2.6).
6. Create a *web_service* element in the Test Case document.

7. Insert the name of the Web Service that was obtained in 4 in the *web_service* element.
8. Create an *address* element in the Test Case document.
9. Insert the address of the Web Service that was obtained in 5 in the *address* element.
10. Get all the *port* elements for the Web Service in 4 (See Fig. 2.6, Table 2.4, and List 2.6).
11. Get all the *binding* elements (See Fig. 2.6, Table 2.4, and List 2.5) for the *port* in 10.
12. Get the *portType* element (See Fig. 2.6 Table 2.4, and List 2.4) for the *binding* element in 11.
13. Get all the *operation* elements (See Fig. 2.6, Table 2.4, and List 2.5) for the *portType* in 12.
14. For each *operation* in 13, extract the *name* attribute of this *operation* from WSDL.
15. Add an *operations* element in the Test Case document to insert all the WSDL operation.
16. Add an *operation_name* element as a sub-element of the operations element in 15.
17. Insert the *operation name* obtained in 13 in the *operation_name* element.
18. Obtain the *input message* (See Fig. 2.6, Table 2.4, and List 2.8) of the operation in 14.
19. Create an *input_message* element in the Test Case document.
20. Insert the *input message* name obtained in 18 to the *input_message* element.

21. Extract all the *part* elements (See Fig. 2.6, Table 2.4, and List 2.8) of the input message in 18.
22. Find the *name* and the *type* attributes (See Fig. 2.6, Table 2.4, and List 2.8) for each *part* in 21.
23. Add a *part_name* element in the resulted document.
24. Insert the *name* attribute obtained in 22 as the text of the *part_name* element.
25. Add a *part_datatype* element in the resulted document.
26. Insert the *type* attribute obtained in 22 as the text of the *part_datatype* element.
27. If the *part_datatype* in 25 is primitive (or derived from primitive) then generate the test cases for this part as described in Section 6.3.2.1.
28. If the *part_datatype* in 25 is user-derived then generate the test cases for this part as described in Section 6.3.2.2.
29. If the *part_datatype* in 25 is complex then generate the test cases for this part as described in Section 6.3.2.2.
30. Extract the *output message* (See Fig. 2.6, Table 2.4, and List 2.4) from WSDL for the operation in 14.
31. Create an *output_message* element in the Test Case document.
32. Insert the *output message* name obtained in 29 to the *output_message* element.
33. Extract all the *part* elements (See Fig. 2.6, Table 2.4, and List 2.8) of the output message in 29.
34. Find the *name* and the *type* attributes (See Fig. 2.6, Table 2.4, and List 2.8) for each *part* in 32.
35. Add a *output_part_name* element in the Test Case document.

36. Insert the *name* attribute obtained in 33 as the text of the *output_part_name* element.
37. Add an *output_part_datatype* element in the Test Case document.
38. Insert the *type* attribute obtained in 33 as the text of the *output_part_datatype* element.

The previous steps can be simplified in the following pseudo code:

Step 1: Get WSDL location.

Step 2: Parse WSDL using DOM

Step 3: Create an XML document to store the generated Test Cases.

Step 4 to Step 9: Obtain the *service name* and the *service address* from the parsed WSDL document and update the Test Case XML document by inserting these data.

Step 10 to Step 13: Get the *port, binding, portType, and operation* elements (See Fig. 2.6).

Step 14 to Step 21: For each WSDL *operation*, find the *operation name* attribute and the *input message*, and the *part* elements of the input message. Then update the Test Case XML document by inserting the data.

Step 22 to Step 26: For each *part*, find the *name* and the *type* attributes. Then update the Test Case XML document by inserting the data.

Step 27 to Step 29: Depending on the *part type* go to the specific test case generator for primitive, user-derived, or complex datatypes.

Step 30 to Step 38: Extract the *output message part* elements. Extract the *name* and *type* for each of these parts. Then update the Test Case XML document by inserting the data.

The previous steps are explained in Fig. 6.4 that shows a general architecture that describes how WSDL document are processed in order to generate test cases in the WS-

Robust tool. Fig. 6.4 shows how the tool extracts the different *operation* elements inside WSDL and then checks the *input message part type* in order to decide which processing to use for test case generation from primitive, user-derived, or complex processing.

6.3.2.1 Primitive or Derived from Primitive Datatypes

WS-Robust uses the following steps to generate the test cases for an *operation* with primitive or derived from primitive *part type* (See Fig. 2.5, Fig. 2.6, and Table 2.4) (Note that the following steps are sub steps of Step 27 of the scenario in Section 6.3.2):

1. Add a *test_cases* element in the Test Case document.
2. Connect to the rules database (generated as described in Section 6.2).
3. Select from the rules table the rules (rows) that have the following properties:
   a. The Web Service Datatype (See Table 5.1) field equals the specific primitive or derived from primitive *part type* that is to generate the test cases.
   b. The WSDL Component (See Table 5.1) equal to XML Schema *part type*.
4. For each retrieved test case (row) from the test cases database do the following:
   a. Add the following elements to the resulted XML document of the test cases:
      I. test_case_id
      II. test_data
      III. expected_output
      IV. quality_assessed1
      V. quality_assessed2
      VI. quality_assessed3

b. Populate these elements by the data from the test cases using the fields: ID, Web Service Test Data, Quality Assessed1, Quality Assessed2, and Quality Assessed3 respectively.

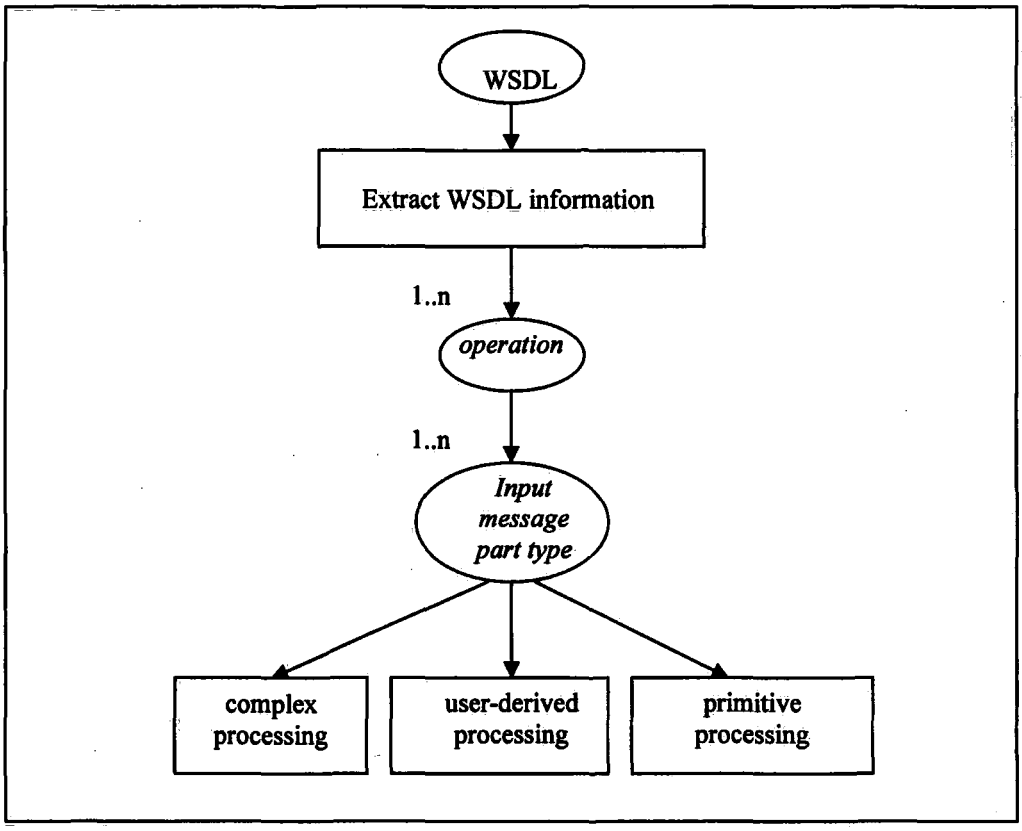


Fig. 6.4 Processing of WSDL Document to Generate Test Cases

The reason of choosing the specific database field in Step 4 from other fields is two fold:

First, these fields can help in creating the Web Service client application that will be used to automatically send SOAP messages to the Web Service under test in order to analyze the response.

Second, these fields are important for the Web Service Requester in order to convey to them the test data used in a certain test case, the expected output, and the quality attributes assessed.

6.3.2.2 User-Derived Datatypes

WS-Robust uses the following steps to generate the test cases for an *operation* with user-derived *part type* (Note that the following steps are sub steps of Step 28 of the scenario in Section 6.3.2):

1. Add a *test_cases* element in the Test Case document.
2. Extract all the simple user-derived datatypes using the WSDL's *types* element (See Fig. 2.6, Table 2.4, and List 2.7).
3. For each simple user-derived *type* extract: datatype *name*, *base* datatype, all the constraining facets with the *value* attribute of each constraint (See List 2.7).
4. For each constraining facet in step 3, apply the following steps (5 to 10).
5. Add the following elements to the resulted document: *part_name, part_datatype, base, facet, value.*
6. Insert the information extracted in Step 3 in the elements of Step 5.
7. Connect to the rules database
8. Select from the rules table (that has the same fields of Fig. 6.1) the rules (rows) that have the following properties:
    a. The WSDL Component (See Table 5.1) field equal to the constraining facet name.
    b. The Web Service Datatype (See Table 5.1) field equal to the base datatype of the constraining facets.

9. from the rules table the rules that have the following properties:
    a. The Web Service Datatype (See Table 5.1) field equal to the base datatype of the constraining facets.
    b. The Test Case ID (See Table 5.1) field has not been used in Step 8.
10. Repeat Step 4 of section 6.3.2.1 if processing primitive or derived from primitive datatypes

The previous steps of test case generation for user-derived datatypes can be summarized in the following:

Step 1: Extract from the rules database the rules with the Web Service Datatype field equal to the base datatype of the constraining facet of the user-derived datatype AND the WSDL Component field equal to the constraining facet name.

Step 2: Extract all the rows with the Web Service Datatype field equal to the base datatype of the constraining facet that have not been selected in Step 1.

Step 1 identifies all the test cases that are based on the constraining facet of the input parameter while Step 2 identifies all the test cases based on the base datatype of the user-derived datatype except the test cases that are already chosen in Step 1.

6.3.2.3 Complex Datatypes

The test cases generation process is easy because complex datatypes consists of a group of primitive and user-derived datatypes, so the test case produced in section 6.3.2.1 and 6.3.2.2 can be used for complex datatypes part. However, as mentioned before, the discussion of complex datatypes will be done in a future work.

6.3.3 Overall Mechanism

The overall mechanism for test case generation is shown in Fig 6.5. It is clear in Fig. 6.5 that:

- Primitive datatype processing depends on the test cases rules
- User derived datatype processing depend on the test cases rules and WSDL's *types* element.
- Complex processing depends on primitive and user-derived processing.

This figure only shows test case generation for Web Services and whether it is the service implementation or the platform being tested depends on each test case rule as defined in the Tables 5.3, 5.4, and 5.6.

6.4 Test Client Generation Mechanism

WS-Robust aims to:

- Generate test cases for a Web Service based on WSDL (Section 6.3)
- Use the test cases to automatically generate SOAP message for the Web Service under test

This Section will discuss how the Test Cases XML document that was generated in Section 6.3 can be used as an input to for a Web Service test client that will automatically generate SOAP message for the Web Service under.

6.4.1 Configuration

To implement the test client generator for a specific test cases document, the following programming language, parser, and Web Services platform.

- Java version 1.5.0_06.
- Document Object Model (DOM) (W3C, 2005) XML parser.
- Axis 1.4

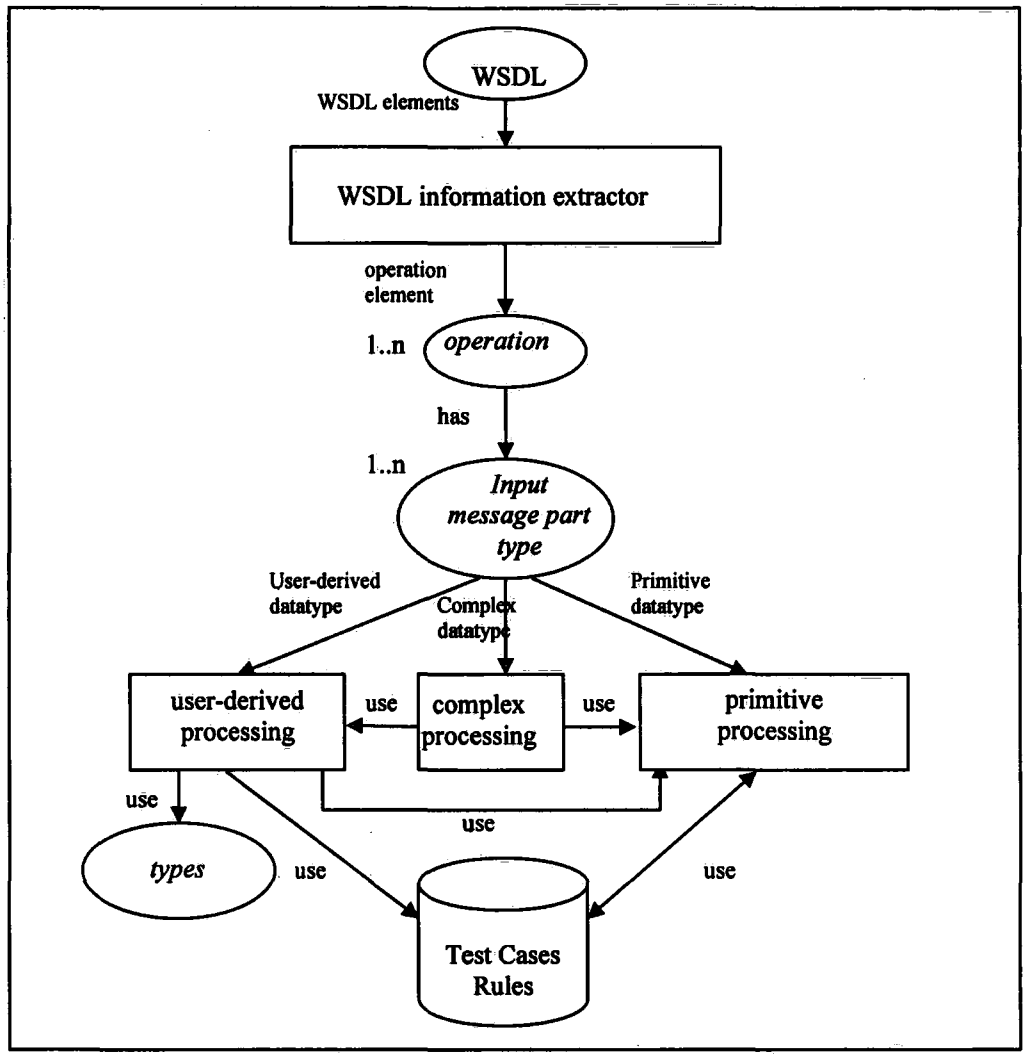


Fig. 6.5. Overall Architecture of Processing WSDL to Generate Test Cases

6.4.2 Scenario

The mechanism for invoking the Web Service under test, with the test data inside the test cases document produced in Section 6.3, is as follows:

1. Parse the XML test cases document generated in Section 6.3 using DOM.
2. Create an XML document to store the test cases of Section 6.3 and the test results. This document will be called XML Responses Document henceforth.
3. Copy the Web Service name element from the resulted document of test cases to the XML Responses Document.
4. Extract the address of the Web Service from the test cases document.
5. For each operation element in the resulted XML document do the following steps (6-9).
6. Invoke the Web Service under test by sending a SOAP *request* (See List 2.10) using the information provided by the address, operation name, test data, and return type elements of the test cases XML document.
7. If the Web Service has responded with a SOAP *response* (See List 2.11) then extract the result of the operation that was invoked and insert it in the responses XML document.
8. If the Web Service under test responded with a SOAP *fault* (See List 2.12) then extract the fault code, fault string, and fault detail elements from the fault message and then insert these elements in the responses XML document.
9. If an operation has more than one part then find the cross product of the parts and then use them to send SOAP messages depending on the information in the XML document of test cases.

The previous steps are used to invoke the Web Service when the operations have parameters of primitive or user-derived datatypes. In case of complex datatypes it is very difficult to automatically generate the test client because this process needs many steps and can not be automated easily. For this reason WS-Robust tool now only handle

generating test client for an operation with a *part* with primitive and user-derived datatypes.

The previous steps are summarized in Fig. 6.6 that describes the mechanism of creating an XML document that contains the test cases for a Web Services together with the actual responses of this Web Service.

Fig 6.6 describes how the address, operation name, test data, and return type that can be extracted from the test cases XML document can be used to generate a test client for the Web Service. The test Client will then generate the test cases with responses document after receiving the SOAP response or SOAP fault from the Web Service under test.

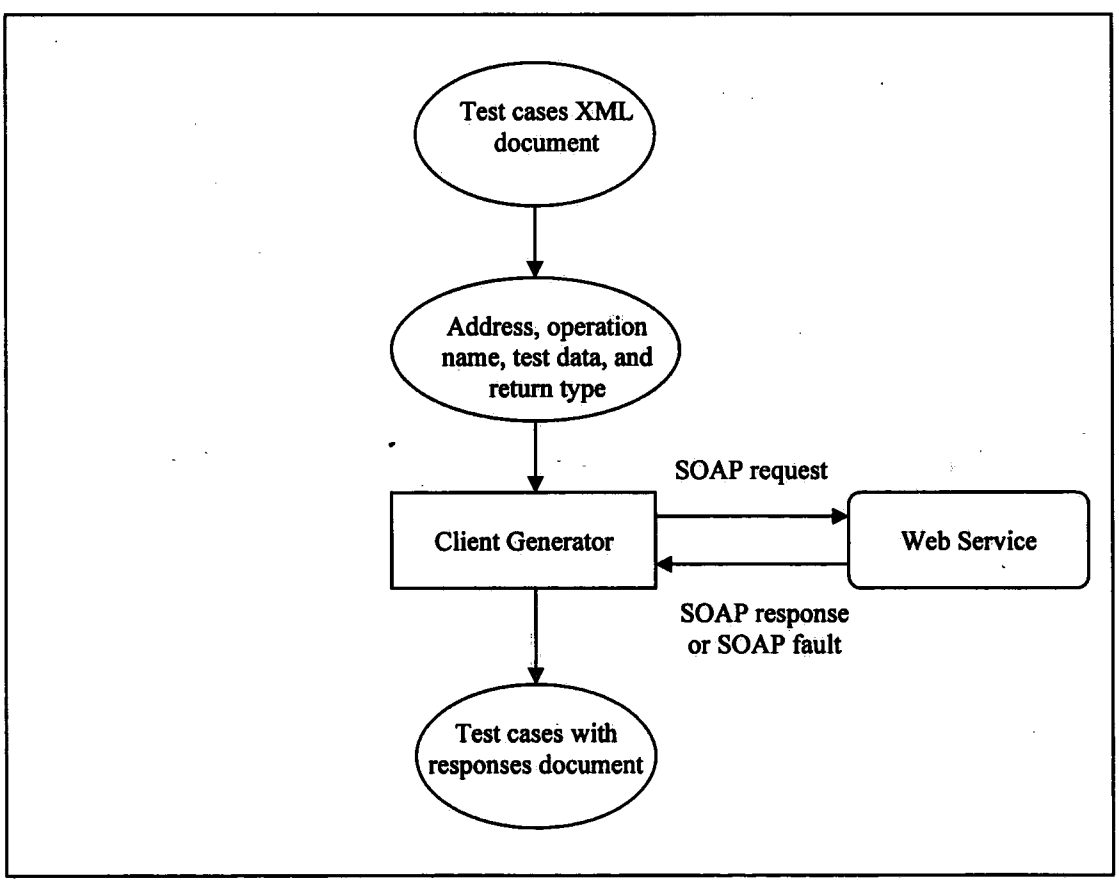


Fig. 6.6. The Mechanism of Generating the Test Cases with Responses Document

6.5 Summary

This Chapter has introduced the implementation details of the WS-Robust tool for Web Service robustness testing. Section 6.2 described how the test cases rules were inserted in a database in order to be used by the components that are responsible for test case generation depending on a specific WSDL document. This Section also described how the test rules can be queried by Web Service Requester or Provider. Section 6.3 described how test cases can be generated depending on WSDL and the test case rules that were inserted in Section 6.2.

Finally, Section 6.4 described how the Web Service under test can be invoked depending on the test cases XML document that was generated in Section 6.3.

This tool can help to increase the trustworthiness of the Web Service Requester because they can check how a Web Service responds to different test cases that were generated based on its interface.

Chapter 7

Evaluation

7.1 Introduction

This chapter evaluates the Web Services robustness testing framework that uses the test case generation rule described in Chapter 5. It does that by using the WS-Robust tool that was implemented in Chapter 6. The framework and WS-Robust will be assessed depending on its ability to test the robustness of Web Services implementation and Web Services platforms such as Axis.

To demonstrate WS-Robust effectiveness, it has been used to assess the robustness of three groups of Web Services example applications:

1. Web Services that accept simple primitive or derived from primitive datatypes as input to its operations (Section 7.2).
2. Web Services that accepts user-derived datatypes as input to its operation (Section 7.3).
3. Commercially available Web Services (Section 7.4).
4. Research based Web Services (Section 7.5).

This chapter will show how the robustness of a Web Service may vary depending on the platform that a Web Service is deployed on (Section 7.5). This will be accomplished by comparing the responses of a Web Service deployed in different platforms, namely, Axis and GLUE.

In the following, mentioning of the WS-Robust tool also implies that the framework of test case generation defined in Chapter 5 is being applied.

7.2 Web Services with Primitive or Derived from Primitive Datatype

The Web Services applications in this section demonstrate that the WS-Robust tool can automatically generate test cases for a Web Service that accepts primitive input *part*, based on WSDL. It also demonstrates that WS-Robust can automatically generate a Web Service test client application that can invoke the Web Service under test using the general test cases and analyze the SOAP response or fault message responses.

7.2.1 Configuration

The examples of this section have been implemented using the following programming language, Web Services platform, and Web server or container:

- Java version 1.5.0_06.
- Axis 1.4
- Apache Tomcat 6.0.

7.2.2 Scenario

Forty one simple Web Services have been implemented and deployed in the Axis Web Service platform which resides on a Tomcat Web server. Each Web Service has an input parameter that has one of the datatypes in Table 5.2. These represents all the primitive or derived from primitive W3C XML Schema datatypes in Fig. 2.5 except the datatypes that are derived by List from other types, namely, ENTITIES, IDREFS, and

NMTOKENS. These datatypes has been excluded because this thesis does not discuss W3C XML Schema List and Union datatypes (W3C, 2004b) (W3C, 2004c).

A Web Service that accepts more than one input part has been implemented in order to use its WSDL to demonstrate the ability of WS-Robust to generate test cases for such Web Services. After deploying these Web Services using Axis, the WSDL document which is automatically generated using Axis, has been used to demonstrate the effectiveness of the test case generation process (see Section 6.3) of WS-Robust. After generating the XML test cases document they have been used to demonstrate the effectiveness of the Web Service test client generation mechanism.

7.2.3 Test Case Generation

Test case generation using WS-Robust will be demonstrated for Web Services that accept single input parameter (Section 7.2.3.1), and for Web Services that accept more the one input parameter (Section 7.2.3.2). The result of the test case generation case studies will be discussed in Section 7.2.3.3.

7.2.3.1 Single Primitive Input Datatype

Test cases were generated for each of the forty one Web Services that have primitive or derived from primitive input parameters using WS-Robust. As an example of the test cases generated for each the forty one Web Services, the generation of test cases for one of these Web Services will be discussed in detail.

The Web Service name is *IntWSService* and it has only one operation called *printInt*. This operation accepts an *int* input and returns a *string* value that represents the *int*

value that has been passed by the SOAP request. Part of the WSDL document of this simple Web Service is shown in List 7.1.

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions
targetNamespace="http://127.0.0.1:8080/axis/IntWS.jws"
.....
   <wsdl:message name="printIntRequest">
      <wsdl:part name="i" type="xsd:int"/>
   </wsdl:message>

   <wsdl:message name="printIntResponse">
      <wsdl:part name="printIntReturn" type="xsd:string"/>
   </wsdl:message>

   <wsdl:portType name="IntWS">

      <wsdl:operation name="printInt" parameterOrder="i">

         <wsdl:input message="impl:printIntRequest"
         name="printIntRequest"/>

         <wsdl:output message="impl:printIntResponse"
         name="printIntResponse"/>

      </wsdl:operation>
   </wsdl:portType>

   <wsdl:binding name="IntWSSoapBinding" type="impl:IntWS">
      .....
   </wsdl:binding>

   <wsdl:service name="IntWSService">
      <wsdl:port binding="impl:IntWSSoapBinding" name="IntWS">
         <wsdlsoap:address
location="http://127.0.0.1:8080/axis/IntWS.jws"/>
      </wsdl:port>
   </wsdl:service>

</wsdl:definitions>
```

List 7.1. WSDL Document for a Web Service that accepts an *int* Input

It is clear in List 7.1 that the *printIntRequest* which is the input message has a part called "*i*" that is of type "*xsd:int*".

The WSDL of List 7.1 was given as input to WS-Robust. Table 7.1 shows the test data that was automatically generated by WS-Robust corresponding to this WSDL.

Table 7.1. Test Data Generated by WS-Robust

ID	WS test data
String_Replacement	rmtpgvezhmoyj
Date-Time_Replacement	2007-12-20
Boolean_Replacement	true
Null_Replacement	null
Max_Value	2147483647
Above_Max	2147483648
Lass_Max	2147483646
Min_Value	-2147483648
Less_Min	-2147483649
Above_Min	-2147483647
Zero_Input	0
Valid_Numeric	12781

WS-Robust produces an XML Test Cases document as mentioned before, List 7.2 shows a portion of the document generated by WS-Robust for this Section Web Service. Similar test cases have been generated using WS-Robust for the other Web Services that accept the other primitive datatypes in Fig. 2.5.

```
<?xml version="1.0" encoding="UTF-8"?>

<web_service>
  <service_name>IntWSService</service_name>
    <address>http://127.0.0.1:8080/axis/IntWS.jws</address>
    <operations>
     <operation>
        <operation_name>printInt</operation_name>
        <input_message>printIntRequest</input_message>
        <ordered_input_parameters>
        <input_part>
           <part_name>i</part_name>
           <part_dataType>int</part_dataType>
           <testings>
            <testing>
             <test_case_id>String_Replacement</test_case_id>
             <test_datatype>String</test_datatype>
             <test_data>rmtpgvezhmoyj</test_data>
             <expected_output>Fault message with proper
            fault string</expected_output>
             <quality_assessed1>Platform robustness
             </quality_assessed1>
             <quality_assessed2>Platform security</quality_assessed2>
             <quality_assessed3>Platform fault tolerance
            </quality_assessed3>
           </testing>
           <testing>
             <test_case_id>Date-Time_Replacemen</test_case_id>
             <test_datatype>Date-Time</test_datatype>
             <test_data>2007-12-20</test_data>
             <expected_output>Fault message with proper fault
            string</expected_output>
             <quality_assessed1>Platform robustness
             </quality_assessed1>
             <quality_assessed2>Platform security</quality_assessed2>
             <quality_assessed3>Platform fault tolerance
             </quality_assessed3>
         </testing>
         <testing>
             <test_case_id>null_Replacement</test_case_id>
             <test_datatype>null</test_datatype>
             <test_data>null</test_data>
             <expected_output>Fault message with proper fault
             string</expected_output>
             <quality_assessed1>Platform
             robustness</quality_assessed1>
             <quality_assessed2>Platform security</quality_assessed2>
             <quality_assessed3>Platform fault
            tolerance</quality_assessed3>
        </testing>
    ....
       </testings>
  ....
</web_service>
```

List 7.2. XML Test Cases Document for a Web Service with *int* Datatype

7.2.3.2 More than one Primitive Input Datatype

All the forty one Web Service accepts only one simple primitive parameter. To show that WS-Robust can handle more that one parameter as an input for a certain operation, a Web Service that accepts two *int* primitive datatype has been implemented and deployed in the Axis platform which resides in a Tomcat Web Server.

Part of the WSDL of this Web Service is shown in List 7.3. It is clear from the WSDL, this Web Service has one operation called *getGreaterNumber* that find the greater between two xsd:*int parts* called *first* and *second* as specified by the *getGreaterNumberRequest* message.

The WSDL in List 7.3 was used as input to WS-Robust and it produced the XML Test Cases document for the Web Service being described. List 7.4 shows part of this XML Test Cases document that was generated using WS-Robust. The approach used by WS-Robust when there is more than one input parameter is to specify the test cases for each parameter separately as clear in List 7.4 where the test cases for the first parameter "*first*" was specified as in the case of single parameter and after that the test cases corresponding to the second input parameter "*second*" were specified. For each of these parameters a test data similar to these described in Table 7.1 are generated.

7.2.3.3 Results

After using WS-Robust the following results have been concluded:

1. WS-Robust is able to generate test cases based on the test cases rules and WSDL for Web Service that has input with any of the W3C XML Schema primitive or derived from primitive datatypes (except the List datatypes).

2. WS-Robust can automatically generate test cases for Web Services that accepts more than one input parameter.

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions
targetNamespace="http://localhost:8080/axis/Greater2.jws"
.....

   <wsdl:message name="getGreaterNumberRequest">
      <wsdl:part name="first" type="xsd:int"/>
      <wsdl:part name="second" type="xsd:int"/>
   </wsdl:message>

   <wsdl:message name="getGreaterNumberResponse">
      <wsdl:part name="getGreaterNumberReturn" type="xsd:int"/>
   </wsdl:message>


   <wsdl:portType name="Greater2">
      <wsdl:operation name="getGreaterNumber" parameterOrder="first
second">
         <wsdl:input message="impl:getGreaterNumberRequest"
name="getGreaterNumberRequest"/>
         <wsdl:output message="impl:getGreaterNumberResponse"
name="getGreaterNumberResponse"/>
      </wsdl:operation>
     </wsdl:portType>
   <wsdl:binding name="Greater2SoapBinding" type="impl:Greater2">
            .....
    </wsdl:binding>

   <wsdl:service name="Greater2Service">
      <wsdl:port binding="impl:Greater2SoapBinding" name="Greater2">
         <wsdlsoap:address
location="http://localhost:8080/axis/Greater2.jws"/>
      </wsdl:port>
   </wsdl:service>

</wsdl:definitions>
```

List 7.3. A WSDL document for a Web Service that accepts two *int* Datatypes

```
<?xml version="1.0" encoding="UTF-8"?>
<web_service>
    <service_name>Greater2Service</service_name>
    <address>http://localhost:8080/axis/Greater2.jws</address>
    <operations>
      <operation>
        <operation_name>getGreaterNumber</operation_name>
        <input_message>getGreaterNumberRequest</input_message>
        <ordered_input_parameters>
           <input_part>
           <part_name>first</part_name>
           <part_dataType>int</part_dataType>
            <testings>
              <testing>
              <test_datatype>String</test_datatype>
              <test_data>dbgvflvmiduqjnhosnoriei</test_data>
               <expected_output>Fault message with proper fault
            string</expected_output>
              <quality_assessed1>Platform
              robustness</quality_assessed1>
              <quality_assessed2>Platform security</quality_assessed2>
              <quality_assessed3>Platform fault
            tolerance</quality_assessed3>
            </testing>
            .........
           <!--More test cases for first part here -->
           </testings>
        </input_part>
           <input_part>
            <part_name>second</part_name>
            <part_dataType>int</part_dataType>
            <testings>
               <testing>
               <test_datatype>Date-Time</test_datatype>
               <test_data>2007-12-06</test_data>
               <expected_output>Fault message with proper fault
            string</expected_output>
               <quality_assessed1>Platform
            robustness</quality_assessed1>
               <quality_assessed2>Platform
            security</quality_assessed2>
               <quality_assessed3>Platform fault
            tolerance</quality_assessed3>
               </testing>
                .........
                <!--More test cases second part here -->
            </testings>
        </input_part>
            .........
</web service>
```

List. 7.4. Test Cases for a Web Service with Two *input* Datatype

7.2.4 Test Client Generation

This section demonstrates that the Test Cases XML document can be used to generate Web Service test client that will invoke the Web Service under test using:

1. The test data in the Test Cases document
2. The Web Service information provided by the Test Cases document such as the Web Service address and the name of the operations.

The examples in this Section use the Axis platform to build the testing client.

Test client generation mechanism using WS-Robust will be demonstrated for a Web Service that accepts single input parameter (Section 7.2.4.1), and for a Web Service that accepts more the one input parameter (Section 7.2.4.2).

7.2.4.1 Single Primitive Input Datatype

The forty one XML Test Cases documents generated in Section 7.2.3.1 have been used as input for the WS-Robust test client generator. An example of the XML document that contains the test cases with the actual response or fault message of the Web Service is given in List 7.5.

Since the test cases depend on the primitive datatype category, namely, Numeric, String, Date-Time, and Boolean (See Table 5.2), each of these datatypes categories will be discussed separately in this section.

```
<?xml version="1.0" encoding="UTF-8"?>
<web_service>
    <service_name>IntWSService</service_name>
    <operations>
        <operation>
            <operation_name>printInt</operation_name>
            <test_cases>
                <test_case>
                    <i>rmtpgvezhmoyj</i>
                    <fault>
                        <fault_code>Server.userException</fault_code>
                        <fault_string>org.xml.sax.SAXException: Bad
                        types (class java.lang.String -&gt; int)
                        </fault_string>
                        <fault_detail>
                            <hostname>e-sci030</hostname>
                        </fault_detail>
                    </fault>
                </test_case>
                <test_case>
                    <i>2007-12-20</i>
                    <fault>
                        <fault_code>Server.userException</fault_code>
                        <fault_string>org.xml.sax.SAXException: Bad
                        types (class java.util.Calendar -&gt; int)
                        </fault_string>
                        <fault_detail>
                            <hostname>e-sci030</hostname>
                        </fault_detail>
                    </fault>
                </test_case>
                <test_case>
                    <i>null</i>
                    <fault>
                        <fault_code>Server.userException</fault_code>
                        <fault_string>No such operation 'printInt'
                        </fault_string>
                        <fault_detail>
                            <hostname>e-sci030</hostname>
                        </fault_detail>
                    </fault>
                </test_case>
                .....
            <!-- More test cases and responses here -->
            </test_cases>
        </operation>
    </operations>
</web_service>
```

List 7.5. Test Cases with Actual Web Service Responses

a) **Numeric Datatypes**

The Numeric Datatypes in Table 5.2 have the following test cases rules from Table 5.3: *String_Replacement, Date-Time_Replacement, Boolean_Replacement, null_Input, Max_Value, Above_Max, Less_Max, Min_Value, Less_Min, Above_Min, Zero_Replacement,* and *Empty_String*. For each of the Numeric Datatypes, Table 7.2 shows the *response* or *fault* messages to each of them according to the experiments that have been conducted.

Table 7.2 uses the following abbreviations:

- FMP: Fault Message with Proper fault string sent by the Web Service platform for changing the datatype of the input *part*.
- RM: Response Message.
- FM: Fault Message.
- NA: Not Applicable.
- null accepted: null has been accepted as input and *response* message has been received by the tool

b) **String Datatypes**

The String Datatypes in Table 5.2 have the following test cases rules in Table 5.3: *Numeric_Replacement, Date-Time_Replacement, Boolean_Replacement, null_Input, Large_String,* and *Empty_String*. For each of the String Datatypes, Table 7.3 shows the *response* or *fault* messages to each of the previous test cases according to the experiments that have been conducted.

c) **Date-Time Datatypes**

The Date-Time Datatypes in Table 5.2 have the following test cases rules from Table 5.3: *Numeric_Replacement, String_Replacement, Boolean_Replacement, null_Input, Empty_String, Valid_Date-Time.* For each of the date-Time Datatypes, Table 7.4 shows the *response* or *fault* messages to each of the previous test cases according to the experiments that have been conducted.

d) **Boolean Datatypes**

The Date-Time Datatypes in Table 5.2 have the following test cases rules from Table 5.3: *Numeric_Replacement, String_Replacement, date-Time_Replacement, null_Input, Empty_String, Valid_Boolean.* For the boolean datatypes which is the only element of Boolean, Table 7.5 shows the *response* or *fault* messages to each of the previous test cases according to the experiments that have been conducted.

Table 7.2 (a). *response* or *fault* messages for the Test Cases with Numeric Datatypes

Test Case / Datatype	String_Replacement	Date-Time_Replacement	Boolean_Replacement	Null_Replacement
decimal	FMP	FMP	FMP	null accepted
integer	FMP	FMP	FMP	null accepted
int	FMP	FMP	FMP	FM with fault string 'No such operation'
byte	FMP	FMP	FMP	FM with fault string 'No such operation'
short	FMP	FMP	FMP	FM with fault string 'No such operation'
long	FMP	FMP	FMP	FM with fault string 'No such operation'
nonPositiveInteger	FMP	FMP	FMP	null accepted
nonNegativeInteger	FMP	FMP	FMP	null accepted
unsignedInt	FMP	FMP	FMP	null accepted
unsignedByte	FMP	FMP	FMP	null accepted
unsignedShort	FMP	FMP	FMP	null accepted
unsignedLong	FMP	FMP	FMP	null accepted
positiveInteger	FMP	FMP	FMP	null accepted
negativeInteger	FMP	FMP	FMP	null accepted
float	FMP	FMP	FMP	FM with fault string 'No such operation'
double	FMP	FMP	FMP	FM with fault string 'No such operation'

Table 7.2 (b)

Test Case / Datatype	Max_Numeric	Above_Max	Less_Max	Min_Numeric
decimal	NA	NA	NA	NA
integer	NA	NA	NA	NA
int	RM	FMP	RM	RM
byte	RM	FMP	RM	RM
short	RM	FMP	RM	RM
long	RM	FMP	RM	RM
nonPositiveInteger	RM	FMP	RM	NA
nonNegativeInteger	NA	NA	NA	RM
unsignedInt	RM	FMP	RM	RM
unsignedByte	RM	FMP	RM	RM
unsignedShort	RM	FMP	RM	RM
unsignedLong	RM	FMP	RM	RM
positiveInteger	RM	FMP	RM	RM
negativeInteger	RM	FMP	RM	RM
float	RM	FMP	RM	RM
double	RM	NA	RM	RM

Table 7.2 (c)

Test Case / Datatype	Above_Min	Less_Min	Zero_Replac-ement	Valid_Numer-ic
decimal	NA	NA	RM	RM
integer	NA	NA	RM	RM
int	RM	FMP	RM	RM
byte	RM	FMP	RM	RM
short	RM	FMP	RM	RM
long	RM	FMP	RM	RM
nonPositiveInteger	NA	NA	RM	RM
nonNegativeInteger	RM	FMP	RM	RM
unsignedInt	RM	FMP	RM	RM
unsignedByte	RM	FMP	RM	RM
unsignedShort	RM	FMP	RM	RM
unsignedLong	RM	FMP	RM	RM
positiveInteger	RM	FMP	RM	RM
negativeInteger	RM	FMP	RM	RM
float	RM	FMP	RM	RM
double	RM	NA	RM	RM

Table 7.3 (a). *response* or *fault* messages for the Test Cases with String Datatypes

Test Case / Datatype	Numeric_Replacement	Date-Time_Replacement	Boolean_Replacement	Null_Replacement
string	FMP	FMP	FMP	null accepted
normalizedString	FMP	FMP	FMP	null accepted
token	FMP	FMP	FMP	null accepted
language	FMP	FMP	FMP	null accepted
Name	FMP	FMP	FMP	null accepted
NMTOKEN	FMP	FMP	FMP	null accepted
NCName	FMP	FMP	FMP	null accepted
ID	FMP	FMP	FMP	null accepted
IDREF	FMP	FMP	FMP	null accepted
Entity	FMP	FMP	FMP	null accepted
base64Binary	FM with fault string: Found character data inside an array element while deserializing	FM with fault string: Found character data inside an array element while deserializing	FM with fault string: Found character data inside an array element while deserializing	null accepted
hexBinary	FMP	FMP	FMP	null accepted
anyURI	FMP	FMP	FMP	null accepted
QName	FMP	FMP	FMP	null accepted
NOTATION	FMP	FMP	FMP	null accepted

Table 7.3 (b)

Test Case / Datatype	Empty_String	Large_String	Valid_String
string	Empty String accepted	RM	RM
normalizedString	Empty String accepted	RM	RM
token	Empty String accepted	RM	RM
language	Empty String accepted	RM	RM
Name	Empty String accepted	RM	RM
NMTOKEN	Empty String accepted	RM	RM
NCName	Empty String accepted	RM	RM
ID	Empty String accepted	RM	RM
IDREF	Empty String accepted	RM	RM
Entity	Empty String accepted	RM	RM
base64Binary	Empty String accepted	RM	RM
hexBinary	Empty String accepted	RM	RM
anyURI	Empty String accepted	RM	RM
QName	Empty String accepted	RM	RM
NOTATION	Empty String accepted	RM	RM

Table 7.4 (a). *response* or *fault* messages for the Test Cases with Date-Time Datatypes

Test Case / Datatype	Numeric_Replacement	String_Replacement	Boolean_Replacement	Null_Replacement
duration	FMP	FMP	FMP	null accepted
dateTime	FMP	FMP	FMP	null accepted
time	FMP	FMP	FMP	null accepted
date	FMP	FMP	FMP	null accepted
gMonthDay	FMP	FMP	FMP	null accepted
gYearMonth	FMP	FMP	FMP	null accepted
gYear	FMP	FMP	FMP	null accepted
gMonth	FMP	FMP	FMP	null accepted
gDay	FMP	FMP	FMP	null accepted

Table 7.4 (b)

Test Case / Datatype	Max_Numeric	Above_Max	Less_Max	Min_Numeric
duration	NA	NA	NA	NA
dateTime	NA	NA	NA	NA
time	NA	NA	NA	NA
date	NA	NA	NA	NA
gMonthDay	RM	FMP	RM	RM
gYearMonth	RM	FMP	RM	RM
gYear	RM	FMP	RM	RM
gMonth	RM	FMP	RM	RM
gDay	RM	FMP	RM	RM

Table 7.4 (c)

Test Case / Datatype	Above_Min	Less_Min	Valid_Date-Time
duration	NA	NA	RM
dateTime	NA	NA	RM
time	NA	NA	RM
date	NA	NA	RM
gMonthDay	RM	FMP	RM
gYearMonth	RM	FMP	RM
gYear	RM	FMP	RM
gMonth	RM	FMP	RM
gDay	RM	FMP	RM

Table 7.5 (a). *response* or *fault* messages for the Test Cases with Boolean Datatypes

Test Case / Datatype	Numeric_Rep-lacement	String_Repla-cement	Date-Time_Rep-lacement	Null_Replace-ment
boolean	FMP	FMP	FMP	null accepted

Table 7.5 (b)

Test Case / Datatype	Empty_String	Valid_Boolean
boolean	FMP	RM

7.2.4.2 More than one Primitive Input Datatype

In the case where there are more than one simple primitive parameter, WS-Robust finds the cross product of the test cases for the parameters of the operation and then uses the result of the cross product to send SOAP messages to the Web Service and analyze its *response* or *fault* message.

To show that WS-Robust can handle automatic client generation for more that one parameter as an input for a certain operation, WS-Robust used the test cases in List 7.4 and generated the XML document in List 7.6 that contains the test cases and their *response* or *fault* message.

Table 7.6 describes the SOAP *response* or *fault* message generated by the Web Service that accepts two *int* parameters (WSDL in List 7.3) when the cross product of the test data of each parameter is used to invoke this Wed Service.

The following abbreviations have been used in table 7.6 in order to fit the results in the table:

S_R: String_Replacement

D_R: date-Time_Replacement

B_R: Boolean_Replacement

N_R: null_Replacement

MX: Max_Value

A_MX: Above_Max

L_Max: Less_Max

MN: Min_Value

L_MN: Less_Min

A_MN: Above_Min

Z_I: Zero_Input

V_N: Valid_Numeric

FMI: fault SOAP message with improper fault string

```
<?xml version="1.0" encoding="UTF-8"?>
<Web_service>
    <service_name>Greater2Service</service_name>
    <operations>
        <operation>
            <operation_name>getGreaterNumber</operation_name>
            <test_cases>
                <test_case>
                    <first>null</first>
                    <second>null</second>
                    <fault>
                        <fault_code>Server.userException</fault_code>
                        <fault_string>No such operation
                        'getGreaterNumber'
                        </fault_string>
                        <fault_detail>
                            <hostname>e-sci030</hostname>
                        </fault_detail>
                    </fault>
                </test_case>
                <test_case>
                    <first>null</first>
                    <second>dbgvflvmiduqjnhosnoriei</second>
                    <fault>
                        <fault_code>Server.userException</fault_code>
                        <fault_string>org.xml.sax.SAXException: Bad
                        types (class java.lang.String -&gt; int)
                        </fault_string>
                        <fault_detail>
                            <hostname>e-sci030</hostname>
                        </fault_detail>
                    </fault>
                <test_case>
                    <first>dbgvflvmiduqjnhosnoriei</first>
                    <second>wjmxbs</second>
                    <fault>
                        <fault_code>Server.userException</fault_code>
                        <fault_string>org.xml.sax.SAXException: Bad
                        types (class java.lang.String -&gt; int)
                        </fault_string>
                        <fault_detail>
                            <hostname>e-sci030</hostname>
                        </fault_detail>
                    </fault>
                </test_case>
      ..............
      <!--More test cases and responses here -->
            </test_cases>
        </operation>
    </operations>
</Web_service>
```

List 7.6. Test Cases and Actual Responses for an Operation with Two Parameters

Table 7.6: Response or fault message for Web Service with Two Input Parameters

Second / First	S_R	D_R	B_R	n_R	MX	A_ MX	L_ MX	MN	L_ MN	A_ MN	Z_I	V_N
S_R	FMP	FMP	FMP	FMP	FMP	FMP	FMP	FMP	FMP	FMP	FMP	FMP
D_R	FMP	FMP	FMP	FMP	FMP	FMP	FMP	FMP	FMP	FMP	FMP	FMP
B_R	FMP	FMP	FMP	FMP	FMP	FMP	FMP	FMP	FMP	FMP	FMP	FMP
n_R	FMP	FMP	FMI	FMI	FMI	FMP	FMI	FMI	FMP	FMI	FMI	FMI
MX	FMP	FMP	FMP	FMI	RM	FMP	RM	RM	FMP	RM	RM	RM
A_MX	FMP	FMP	FMP	FMP	FMP	FMP	FMP	FMP	FMP	FMP	FMP	FMP
L_MX	FMP	FMP	FMP	FMI	RM	FMP	RM	RM	FMP	RM	RM	RM
MN	FMP	FMP	FMP	FMI	RM	FMP	RM	RM	FMP	RM	RM	RM
L_MN	FMP	FMP	FMP	FMP	FMP	FMP	FMP	FMP	FMP	FMP	FMP	FMP
A_MN	FMP	FMP	FMP	FMI	RM	FMP	RM	RM	FMP	RM	RM	RM
Z_I	FMP	FMP	FMP	FMP	RM	FMP	RM	RM	FMP	RM	RM	RM
V_N	FMP	FMP	FMP	FMP	RM	FMP	RM	RM	FMP	RM	RM	RM

7.2.4.3 Results

After analyzing Table 7.2, Table 7.3, Table 7.4, Table 7.5, Table 7.6, List 7.4, and List 7.5, the following results can be concluded:

- WS-Robust can automatically generate a test client depending on the XML test cases document generated in Section 7.2.3.
- For each test case in the test cases document, WS-Robust specifies the *output* or *fault* message details (List 7.3).

- WS-Robust can analyze the fault messages, in the resulted XML document of the test cases and their response, by specifying the *fault code*, *fault string*, and the *fault detail*. (See List 2.12 and List 7.3).
- The *null_Replacement* test case in Table 7.2 revealed a robustness failure in the Axis Web Service platform because this platform has accepted the *null* value as an input when the input parameter for all the XML Schema datatypes except *int, byte, short, long, float, and double*. So Axis in not consistent in handling the null input.
- Axis produced a robustness failure when returning the fault string in the fault message when rejecting the *null* input in the case of *int, byte, short, long, float, and double* datatypes. The fault message was "No such operation" while the operation existed in the Web Service. The fault message should have been for example *"can not accept a null value because the input parameter of type int"*.
- Table 7.2 showed that the Web Services implementations, for the Web Services that accept *decimal, integer, nonPositiveInteger, nonNegativeInteger, unsignedInt, unsignedByte, unsignedShort, positiveInteger,* and *negativeInteger* produced a robustness failure when applying the Null_Replacement test cases generation rule because they did not send a SOAP fault that contains a proper fault response.
- Table 7.2 showed that Axis platform was robust when applying the *String_Replacement, Date-Time_Replacement, Boolean_Replacement, Above_Max,* and *Less_Min* test cases rules for the Numeric XML Schema datatypes because Axis always returned a SOAP fault with proper fault string describing the fault.
- Table 7.2 showed that the Web Service implementations, of the Web Services that expect XML Schema Numeric datatypes as input, are robust when applying the

Max_Numeric, Less_Max, Min_Numeric, Above_Min, and *Zero_Replacement* since those extreme values did not cause any problems to the Web Service implementation and a SOAP response has been sent to WS-Robust.

- Table 7.3 showed that Axis platform is robust when using the *Numeric_Replacement, Date-Time_Replacement,* and *Boolean_Replacement* test cases rules with String datatypes except when the input to a Web Service operation has *base64Binary* datatype. The reason for this, is that, a SOAP fault message with a proper fault string has been produced by Axis when applying these test cases rules, for the datatype *base64Binary*, the SOAP fault contained the fault string *"Found character data inside an array element while deserializing"* which does not describe the fault that has happened.
- Table 7.3 showed that the Web Services implementations that expect String datatypes are robust when applying the *Large_String* and *Valid_String* test case. The reason for this is that the Web Services implementations returned a SOAP response and the input suggested in the test cases rules did not cause the Web Services to behave improperly.
- Table 7.3 showed that the Web Services implementations that expect String datatypes are not robust when applying the *Empty_String* test case generation rule. The reason for that is the Web Service implementation did not return a SOAP fault with a proper fault string.
- Table 7.3 showed that the Axis is not robust when applying the *null_Replacement* test case generation rule with String datatypes. The reason is that Axis accepted the null input and did not generate a SOAP fault with proper fault string.

- Table 7.4 shows that Axis platform is robust when using the *Numeric_Replacement, String_Replacemen, Boolean_Replacemnt, Above_Max, and Less_Min* test cases rules with Date-Time datatypes (See table 5.2). The reason for this is that a SOAP fault message with a proper fault string has been produced by Axis when applying these test cases rules.
- Table 7.4 shows a robustness failure in the Axis Web Service platform because this platform has accepted the *null* value as an input when the input parameter has any of Date-Time XML Schema datatypes.
- Table 7.4 showed that the Web Services that accepts *Date-Time* produced a robustness failure when applying the *Null_Replacement* test cases generation rule because they did not send a SOAP fault that contains a proper fault response.
- Table 7.4 showed that the Web Services implementations that expect Date-Time datatypes are robust when applying the *Max_Numeric, Less_Max, Min_Numertic, Above_Min* and *Valid_Date-Time* test cases. The reason for this is that the Web Services implementations returned a SOAP response.
- Table 7.5 showed that the Web Services that accepts *Boolean* produced a robustness failure when applying the *Null_Replacement* test cases generation rule because they did not send a SOAP fault that contains a proper fault response.
- Table 7.5 showed that the Axis was robust when applying the *Numeric_Replacement, String_Replacement, Date-Time_Replacement,* and *Empty_String* test cases. The reason for this is that the Axis responded with a SOAP fault with a proper fault string as expected.

- Table 7.5 showed that the Web Services implementations that expect Boolean datatypes are robust when applying the *Valid_Boolean* test case. The reason for this is that the Web Services implementations returned a SOAP response.
- Table 7.6 showed that when using the cross product some faults may hide the other faults in case of more than one input parameter. For example, in the case when using *String_replacement* test case with the *first* parameter and the *null_Replacement* for the second parameter, then Axis sent a SOAP fault with proper fault string. While when using the *Max_Value* with the first parameter (which gave a response SOAP message in the case of single *int* parameter in Table 7.2) and the null_Replacement with the *second* parameter, then Axis responded with a fault string with improper fault string *"no such operation"*. This means that in the first case, the fault *"no such operation"* was hidden because the input contains another fault which is the string replacement.
- It is noticed in Table 7.6 that the rows are identical to the columns. For this reason and the discussion of the previous step, and also to reduce the number of test cases, it is better that when invoking the Web Service to have the test cases of the faulty input applied to only one parameter and all the other parameters must be given a valid input so that the faults are not hidden and also getting lest test cases and consequently reducing the cost of testing without sacrificing the precision of the robustness estimate.
- WS-Robust did not reveal any security faults in Axis platform when the input parameter is primitive or derived from primitive because, for all the SOAP faults in Table 7.1 through Table 7.6, the detailed element of the fault message contained one sub-element which was the name of the host where the fault has

occurred. This means that no stack trace was provided with the fault message that could be used by malicious Service Requesters to harm a Web Service.

To summarize the result obtained, Table 7.7 shows the numbers of Test Cases, the number of robustness failures detected in both the Web Services implementation (WS Failures) and the Axis platform, for each of the forty one example Web Services that accepts different XML Schema datatype.

As Table 7.7 shows, 359 SOAP request Test Cases were used to assess the robustness of the 41 Web Services of Section 7.2. These test cases were able to detect 50 robustness failures in the Web Services implementations (WS Failures) and 44 robustness failures in the Axis Web Services platform (Axis Failures).

Table 7.7. Implementation and Platform Robustness Failure for the Web Services Examples

Web Service Input Datatype	Test Cases	WS Failures	Axis Failures	Fault Description
decimal	6	1	1	Handling null input
integer	6	1	1	Handling null input
int	12	0	1	Handling null input
byte	12	0	1	Handling null input
short	12	0	1	Handling null input
long	12	0	1	Handling null input
nonPositiveInteger	9	1	1	Handling null input
nonNegativeInteger	9	1	1	Handling null input
unsignedInt	12	1	1	Handling null input
unsignedByte	12	1	1	Handling null input
unsignedShort	12	1	1	Handling null input
unsignedLong	12	1	1	Handling null input
positiveInteger	12	1	1	Handling null input
negativeInteger	12	1	1	Handling null input
float	12	0	1	Handling null input
double	10	0	1	Handling null input
string	7	2	1	Handling null input and empty string
normalizedString	7	2	1	Handling null input and empty string
token	7	2	1	Handling null input and empty string
language	7	2	1	Handling null input and empty string
Name	7	2	1	Handling null input and empty string
NMTOKEN	7	2	1	Handling null input and empty string
NCName	7	2	1	Handling null input and empty string
ID	7	2	1	Handling null input and empty string
IDREF	7	2	1	Handling null input and empty string
Entity	7	2	1	Handling null input and empty string
base64Binary	7	2	4	Handling datatype replacement, handling null input and handling empty string
hexBinary	7	2	1	Handling null input and empty string
anyURI	7	2	1	Handling null input and empty string
QName	7	2	1	Handling null input and empty string
NOTATION	7	2	1	Handling null input and empty string
duration	5	1	1	Handling null input
dateTime	5	1	1	Handling null input
time	5	1	1	Handling null input
date	5	1	1	Handling null input
gMonthDay	11	1	1	Handling null input
gYearMonth	11	1	1	Handling null input
gYear	11	1	1	Handling null input
gMonth	11	1	1	Handling null input
gDay	11	1	1	Handling null input
boolean	7	1	1	Handling null input
Total	**359**	**50**	**44**	

7.3 Web Services with User-derived Datatype

This Section demonstrates that WS-Robust tool can automatically generate test cases for the Web Service that accepts a used-derived datatype and automatically generate a Web Service test client application to test these Web Services.

7.3.1 Configuration

To implement the examples of this Section, the same programming language, Web Services platform, and Web server or container of Section 7.2 has been used. Namely:

- Java version 1.5.0_06.
- Axis 1.4
- Apache Tomcat 6.0.

7.3.2 Scenario

A Web Service that accepts a user-derived datatype has been implemented, the *types* element of the WSDL of this Web Service is given in List 7.5. This Web Service accepts an *integer part* as input and this *part* has the *type* (See Fig 2.6) *moreFiveType* (See List 7.7). This datatype has *minInclusiv* and *maxInclusive*. This Web Service has been used to demonstrate the ability of WS-Robust to generate test cases (Section 7.3.3) and test client (Section 7.3.4) for a Web Service with a user-derived datatype.

The WSDL for this Web Service is similar to the WSDL in List 7.1 but with the added types element in List 7.7 in order to describe the user-derived datatype moreFiveType.

```
<types>
  <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   targetNamespace= "http://localhost:8080/axis/IntegerWS.jws">
     <xsd:simpleType name="moreFiveType">
        <xsd:restriction base="xsd:integer">
              <xsd:minInclusive value = "5"/>
              <xsd:maxInclusive value = "100"/>
        </xsd:restriction>
     </xsd:simpleType>
   </xsd:schema>
</types>
```

List 7.7 WSDL *types* element that Contains a User Derived Datatype

7.3.3 Test Case Generation

Test cases generation using WS-Robust will be demonstrated using the Web Service described in Section 7.3.2. As in the case of primitive datatypes, test case generation will depend on the WSDL document for the Web Service. However, in case of user-derived *part*, the *types* element inside WSDL must be analyzed to determine which test cases to use depending on the rules of Table 5.6.

List 7.8 shows the XML test cases document that has been generated automatically by WS-Robust. The resulted XML document of the test cases specifies for each operation inside WSDL the input parameters of the request message and their parts, for the user-derived part it specifies the base datatype, the facet name, and the value of this facet together with the test cases that can be obtained from the test cases database based on the specific constrains, its base datatype and its value.

```
<?xml version="1.0" encoding="UTF-8"?>

<web_service>
  <service_name>IntegerMinInclusiveMaxInclusiveService</service_name>
    <address>http://127.0.0.1:8080/axis/IntegerWS.jws</address>
    <operations>
        <operation>
            <operation_name>printInteger</operation_name>
            <input_message>printIntegerRequest</input_message>
            <ordered_input_parameters>
                <input_part>
                    <part_name>integer1</part_name>
                    <part_dataType>moreFiveType</part_dataType>
                    <base>integer</base>
                    <facet>minInclusive</facet>
                    <value>5</value>
                    <test_cases>
                        <test_case>
                            <test_case_id>Min_Value</test_case_id>
                            <test_datatype>integer</test_datatype>
                            <test_data>5</test_data>
                            <expected_output>Response
                        message</expected_output>
                            <quality_assessed1>WS implementation
                            robustness</quality_assessed1>
                            <quality_assessed2>WS implementation
                            security</quality_assessed2>
                        </test_case>
                       <!-- ..... more test cases here -->
                    </test_cases>
                    <facet>maxInclusive</facet>
                    <value>100</value>
                    <test_cases>
                        <test_case>
                            <test_case_id>Max_Value</test_case_id>
                            <test_datatype>integer</test_datatype>
                            <test_data>100</test_data>
                            <expected_output>Response
                        message</expected_output>
                            <quality_assessed1>WS implementation
                            robustness</quality_assessed1>
                            <quality_assessed2>WS implementation
                            security</quality_assessed2>
                        </test_case>
                        <!-- ..... more test cases here -->
                    </test_cases>
                </input_part>
            </ordered_input_parameters>
            <!-- output part description here -->
          </operation>
    </operations>
</web_service>
```

List 7.8. Test Cases for a Web Service with a User-Derived input datatype

7.3.4 Test Client Generation

The Web Service test client uses the Axis platform as in the case of primitive datatypes. List 7.9 shows the test cases and the actual responses that were generated automatically by WS-Robust.

WS-Robust does not only use the test cases that are provided in the XML test cases document (See List 7.8) but also uses the test cases for the base datatype which is *integer* in List 7.8. However, if a test case is used by the base datatype (which is primitive or derived from primitive) and the user-derived (based on a constraining facet), then the test case of the user-derived is the only one that will be used for it overrides the test case for the base primitive datatype.

For example, in List 7.8, the *Min_Value* test case generation rule is used by the base datatype which is *integer* and also used by the *minInclusive* constraining facet that is used with the *moreFiveType* (See List 7.7). In this case, *Min_Value* which is used with the *minInclusive* facet will be used when generating test cases for a parameter with *moreFiveType*. The reason for this is the minimum value is already constrainted by the *minInclusive* constraining facet so we should not use the minimum default value for

```
<?xml version="1.0" encoding="UTF-8"?>

<web_service>IntegerMinInclusiveMaxInclusiveService</service_name>
    <operations>
        <operation>
            <operation_name>printInteger</operation_name>
            <test_cases>
                <test_case>
                    <integer1>tvleyohzfrbip</integer1>
                    <fault>
                        <fault_code>Server.userException</fault_code>
                        <fault_string>org.xml.sax.SAXException: Bad
                        types (class java.lang.String -&gt; class
                        java.math.BigInteger)
                        </fault_string>
                        <fault_detail>
                            <hostname>e-sci030</hostname>
                        </fault_detail>
                    </fault>
                </test_case>
                <test_case>
                    <integer1>2008-01-11</integer1>
                    <fault>
                        <fault_code>Server.userException</fault_code>
                        <fault_string>org.xml.sax.SAXException: Bad
                        types (class java.util.Calendar -&gt; class
                        java.math.BigInteger)</fault_string>
                        <fault_detail>
                            <hostname>e-sci030</hostname>
                        </fault_detail>
                    </fault>
                </test_case>
                 .....
                <!-more test cases and responses for integer here -->
                <!-test cases for minInclusive starts here -->
                <test_case>
                    <integer1>5</integer1>
                    <output>The integer passed is 5</output>
                    <invokation_time>16 ms</invokation_time>
                </test_case>
                <test_case>
                    <integer1>6</integer1>
                    <output>The integer passed is 6</output>
                    <invokation_time>16 ms</invokation_time>
                </test_case>
                <test_case>
                    <integer1>4</integer1>
                    <output>The integer passed is 4</output>
                    <invokation_time>15 ms</invokation_time>
                </test_case>
              <!--test cases for other constraints here -->
            </test_cases>
        </operation>
    </operations>
</web_service>
```

List 7.9. Test Cases with Responses for User-derived Datatype

integer datatype. In the other hand, if the *minInclusive* constraint is not specified for the *moreFiveType* then the minimum value of *integer* would be used instead.

Table 7.8 will summarize the SOAP *response* or *fault* message for each combination of constraining facet and test cases used for the *moreFiveType* (See List 7.5). Table 7.8 shows only the test cases that are based on the constraining facet but not the test case that are the based on the primitive datatype (*integer*) since these test case has been discussed in Section 7.2.

Table 7.8 (a) SOAP *response* or *fault* messages for test cases for Numeric Boundaries Constraints

Test Case / Constraining Facet	Min_Value	Above_Min	Less_Min
minInclusive	RM	RM	RM

Table 7.8 (b)

Test Case / Constraining Facet	Max_Value	Above_Max	Less_Max
maxInclusive	RM	RM	RM

7.3.5 Results

Section 7.3.3 and Section 7.3.4 provided the following results:

1. WS-Robust is able to automatically generate test cases for a Web Service with a user-derived input part based on analyzing WSDL's *types* element and the test case generation rules.

2. WS-Robust can automatically generate test case client that invokes the Web Service under test based on the test cases obtained of step 1.
3. Table 7.7 shows that the Web Service platform (Axis) and the Web Service implementation for the Web Service used as an example in this section is not robust because they both accepted the *Less_Min* test case input and the *Above_Max* test case input and did not generate a fault message with a proper fault string.
4. The Axis platform does not have the facility to check if the constraints in the WSDL's *types* element are satisfied in the SOAP request because in the Less_Min and Above_Max axis did not generate any SOAP fault.
5. Adding more constraints to the input parameter increases the testability of the Web Services and increases the detected robustness failures.

7.4 Testing a Commercial Web Services

To demonstrate the effectiveness of the approach in this thesis to detect robustness fault in Web Services, a real commercial Web Service is tested. The Web Service chosen is the Amazon Web Service (provided by http://www.amazon.com). The input to the Amazon is of complex datatype which is beyond the scope of this thesis. However, the test cases rules have been manually applied to generate a SOAP test request to Amazon to assess its robustness.

7.4.1 Scenario

To build a client to the Amazon Web Service the *wsdl2jave* program provided by Axis was used with the Amazon WSDL document. *Wsdl2jave* produced fifty one java files that are needed to write clients to access the information provided by Amazon. List 7.10 gives a small portion of the *types* element of the Amazon WSDL that contains a complex datatype that represent an ASIN request datatype. ASIN stand for Amazon Standard Identification Number which is a unique number given to each Amazon product, for the books, ASIN is simply the ISBN of the book.

In order to apply this thesis' test cases rules discussed in Chapter 5 to the Amazon Web Service, a Web application that represents a client to the Amazon Web Service has been implemented. This client accepts an ASIN complex input and then invokes the Amazon Web Service to get the details of the item with this ASIN.

```
<xsd:complexType name="AsinRequest">
 <xsd:all>
  <xsd:element name="asin" type="xsd:string"/>
  <xsd:element name="tag" type="xsd:string"/>
  <xsd:element name="type" type="xsd:string"/>
  <xsd:element name="devtag" type="xsd:string"/>
  <xsd:element name="offer" type="xsd:string" minOccurs="0"/>
  <xsd:element name="offerpage" type="xsd:string" minOccurs="0"/>
  <xsd:element name="locale" type="xsd:string" minOccurs="0"/>
 </xsd:all>
</xsd:complexType>
```

List 7.10. A Complex Datatype that Represent ASIN Request

7.4.2 Test Case Generation

When analyzing the WSDL document of the Amazon Web Service, the following add difficulty to the testing process:

- The only datatype that is used for all the input parameters is the *string* datatype.
- There is no constraining facet on any of the datatypes, which mean only primitive is used.

These two points have been noticed in most of the Web Services that has been analyzed on the http://www.xmethods.net site. So to increase the testability of Web Services, the Web Service Providers must use the appropriate datatypes for the different elements or parameters of their Web Services and they must also add more specifications or constrains to these parameters.

This poor commercial datatyping problem must be addressed by the Service Providers in order to increase the trustworthiness; because:

- The Service Requester will have more understanding of the Web Service being described by WSDL.
- The Service Requester will know what the constraints on the input parameters are.

Since all the elements of the ASIN datatypes are of primitive *string* datatype, the test cases in Table 5.3 (*Numeric_Replacement, Date-Time_Replacement, Boolean_Replacement, null_Replacement, Large_String, and Empty_String*) will be used in order to analyze the *robustness of the Service*. Table 7.8 gives the results of applying these test cases for the *AsinRequest* datatype. The rows of Table 7.8 represent the test case generation rules for a string datatype and the columns represent the elements of the AsinRequest complex datatype. To apply the test cases for a specific AsinRequest element (column), the other elements will be given a valid value and then the value of this specific element will be changed in the SOAP request depending on the test case (row).

7.4.3 Results

The results are:

- The Amazon WSDL uses the *string* datatype for all the input and output parameters described in the WSDL and this will reduce the testability of the Amazon Web Service.
- The Amazon WSDL uses only the primitive datatypes (*string*) and does not add any constraining facets to the input or output parameters of the operations and this will also decrease the testability. The only constraint id minOccurence that specifies the times that an element can occur.

For Table 7.9, the results of each column will be given separately as follows:

1) **ASIN**

- Amazon Web Service is robust for the all the test cases.
- The Amazon Web Service is not secure when the *null_Replacement* test case is applied because the SOAP fault contained a stack trace that may be used to harm this Web Service.

2) **Tag**

- Amazon Web Service is not robust when applying the *Numeric_Replacement, Date-Time_Replacement, Boolena_Replacement,* and *Empty_String* because it returned a response SOAP message while a fault message with proper fault string was expected for these test cases.
- The Amazon Web Service is not secure when the *null_Replacement* test case is applied for the same reason mentioned for the ASIN element.

Table 7.9 (a). Amazon *response* or *fault* messages for String Test Cases

AsinRequest / Test Case	ASIN	Tag	Type	devtag
Numeric_Replacement	FMP	RM	FM with improper* fault string and stack trace	FM with improper fault string and stack trace
Date-Time_Replacement	FMP	RM	FM with improper fault string and stack trace	FM with improper fault string and stack trace
Boolean_Replacement	FMP	RM	FM with improper fault string and stack trace	FM with improper fault string and stack trace
null_Replacement	FMP and stack trace	FMP and stack trace	FMP and stack trace	FM with improper fault string and stack trace
Large_String	RM	RM	FM with improper fault string and stack trace	RM
Empty_String	FMP	RM	FM with improper fault string and stack trace	FM with improper fault string and stack trace

Table 7.9 (b)

AsinRequest / Test Case	Offer	Offerpage	Locale
Numeric_Replacement	RM	RM	RM with US dollar sign
Date-Time_Replacement	RM	RM	RM with US dollar sign
Boolean_Replacement	RM	RM	RM with US dollar sign
null_Replacement	RM	RM	RM with US dollar sign
Large_String	RM	RM	RM
Empty_String	RM	RM	RM with US dollar sign

* the SOAP fault with the stack trace is given in List 7.11.

```
AxisFault
faultCode: {http://schemas.xmlsoap.org/soap/envelope/}Server.userException
faultSubcode:
faultString: org.w3c.dom.DOMException: WRONG_DOCUMENT_ERR: A node is
used in a different document than the one that created it.
faultActor:
faultNode:
faultDetail:
    {http://xml.apache.org/axis/}stackTrace:org.w3c.dom.DOMException: WRONG_
DOCUMENT_ERR: A node is used in a different document than the one that created it.
    at org.apache.xerces.dom.ParentNode.internalInsertBefore(Unknown Source)

    at org.apache.xerces.dom.ParentNode.insertBefore(Unknown Source)
    at org.apache.xerces.dom.NodeImpl.appendChild(Unknown Source)
    at org.apache.axis.message.SOAPFaultBuilder.onEndChild(SOAPFaultBuilder.
java:305)
    at org.apache.axis.encoding.DeserializationContext.endElement(Deserializ
ationContext.java:1090)
    at org.apache.xerces.parsers.AbstractSAXParser.endElement(Unknown Source)
  at org.apache.xerces.impl.XMLNSDocumentScannerImpl.scanEndElement(Unknown
Source)
    at org.apache.xerces.impl.XMLDocumentFragmentScannerImpl$FragmentContent
Dispatcher.dispatch(Unknown Source)
    at
org.apache.xerces.impl.XMLDocumentFragmentScannerImpl.scanDocument(Unknown
Source) at org.apache.xerces.parsers.XML11Configuration.parse(Unknown Source)
 at org.apache.xerces.parsers.XML11Configuration.parse(Unknown Source)
    at org.apache.xerces.parsers.XMLParser.parse(Unknown Source)
    at org.apache.xerces.parsers.AbstractSAXParser.parse(Unknown Source)
    at org.apache.xerces.jaxp.SAXParserImpl$JAXPSAXParser.parse(Unknown
Source)
    at org.apache.xerces.jaxp.SAXParserImpl.parse(Unknown Source)
    at org.apache.axis.encoding.DeserializationContext.parse(Deserialization
Context.java:227)

.....
```

List 7.11. A SOAP fault with Improper fault string and stack trace

3) **Type**

- The Amazon Web Service was not robust when applying the *Numeric_Replacement, Date-Time_Replacement, Boolean_Replacement,* and *Empty_String* because it returned a SOAP fault message with the improper fault message: *"org.w3c.dom.DOMException: WRONG_DOCUMENT_ERR: A node is used in a different document than the one that created it."* (See List 7.9)
- The Amazon Web Service is not secure when applying each of the test cases generation rules because it returned a stack trace in the SOAP fault.
- The Amazon Web Service is not robust when applying the *Large_String* test cases generation rule because it returned a fault message while a SOAP response was expected.

4) **devtag**

- The Amazon Web Service was not robust when applying the *Numeric_Replacement, Date-Time_Replacement, Boolean_Replacement,* and *Empty_String* because it returned a SOAP fault message with the improper fault message.
- The Amazon Web Service is not secure when applying *Numeric_Replacement, Date-Time_Replacement, Boolean_Replacement, null_Replacement* and *Empty_String* test cases generation rules because it returned a stack trace in the SOAP fault.
- The Amazon Web Service was robust when applying the *Large_String* test case.

5) **Offer**

- The Amazon Web Service was not robust when applying the *Numeric_Replacement, Date-Time_Replacement, Boolean_Replacement,*

null_Replacement and *Empty_String* because it returned a SOAP response while a fault message with the proper fault message was expected by these test cases.

- The Amazon Web Service was robust when applying the *Large_String* test case.

6) **Offerpage**

The results obtained when applying the test cases generation rules to the Offerpage element is the same as those obtained for the Offer element.

7) **Locale**

- The Amazon Web Service is not robust when applying the Numeric_Replacement, Date-Time_Replacement, Boolean_Replacement, null_Replacement and Empty_String test cases because Amazon used the default value for the locale parameter, which the US dollar and returned a SOAP response when A SOAP fault with proper fault string was expected.

In summary, the previous results demonstrated that the rules for test case generation for Web Services proposed in this thesis can detect robustness faults in a real commercial Web Services.

7.5 Testing a Research-based Web Service

All the Web Services examples so far, except for the Amazon, were local Web Services that are deployed in the same computer as the WS-Robust tool. This Section will demonstrate the ability of WS-Robust tool to test remote Web Services.

7.5.1 Configuration

This Section examples use WS-Robust tool and the Axis platform.

7.5.2 Scenario

WS-Robust was used with a remote Web Services that is used to find the square root of the input parameter. This Web Service is deployed in a server called e-sci035 that belongs to Durham University. The WSDL of this Web Service is described in List 7.12.

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="urn:calculation"
xmlns:apachesoap="http://xml.apache.org/xml-soap"
xmlns:impl="urn:calculation" xmlns:intf="urn:calculation"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<!--WSDL created by Apache Axis version: 1.3
Built on Oct 05, 2005 (05:23:37 EDT)-->
   <wsdl:message name="squareRootResponse">
      <wsdl:part name="squareRootReturn" type="xsd:double"/>
   </wsdl:message>
   <wsdl:message name="squareRootRequest">
      <wsdl:part name="in0" type="xsd:double"/>
   </wsdl:message>
   <wsdl:portType name="SquareRoot">
      <wsdl:operation name="squareRoot" parameterOrder="in0">
         <wsdl:input message="impl:squareRootRequest"
      name="squareRootRequest"/>
         <wsdl:output message="impl:squareRootResponse"
      name="squareRootResponse"/>
      </wsdl:operation>
   </wsdl:portType>
   <wsdl:binding name="SquareRootSoapBinding" type="impl:SquareRoot">
      <wsdlsoap:binding style="rpc"
      transport="http://schemas.xmlsoap.org/soap/http"/>
      <wsdl:operation name="squareRoot">
         <wsdlsoap:operation soapAction=""/>
         <wsdl:input name="squareRootRequest">
         <wsdlsoap:body
      encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
      namespace="urn:calculation" use="encoded"/>
         </wsdl:input>
         <wsdl:output name="squareRootResponse">
         <wsdlsoap:body
      encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
      namespace="urn:calculation" use="encoded"/>
         </wsdl:output>
      </wsdl:operation>
   </wsdl:binding>
   <wsdl:service name="SquareRootService">
      <wsdl:port binding="impl:SquareRootSoapBinding"
      name="SquareRoot">
      <wsdlsoap:address
      location="http://
      e-sci035.dur.ac.uk:8080/axis/services/SquareRoot"/>
      </wsdl:port>
   </wsdl:service>
</wsdl:definitions>
```

List 7.12.WSDL Document of the Square Root Web Service

7.5.3 Test Case and Test Client Generation

The WSDL in List 7.12 was used as an input to the WS-Robust, part of the test cases generated shown in List 7.13, and the responses document in List 7.14.

Table 7.10 summarizes all the test data and the responses from the square root Web Service. It can be concluded from this table that the only robustness failure occurred with the null replacement input where it responded with a fault message with an improper fault string (FMI).

The result of this section is that the WS-Robust is able to assess the robustness of Web Services that are deployed on a systems remote from the WS-Robust tool and written and implemented by a third party.

```
<?xml version="1.0" encoding="UTF-8"?>
<web_service>
    <service_name>SquareRootService</service_name>
    <address>
      http://e-sci035.dur.ac.uk:8080/axis/services/SquareRoot
    </address>
    <operations>
      <operation>
        <operation_name>squareRoot</operation_name>
        <input_message>squareRootRequest</input_message>
        <ordered_input_parameters>
           <input_part>
               <part_name>in0</part_name>
               <part_dataType>double</part_dataType>
               <test_cases>
                 <test_case>
                 <test_case_id>String_Replacement</test_case_id>
                 <test_datatype>String</test_datatype>
                 <test_data>oilcqhiflyzzasgkzcplg</test_data>
                  <expected_output>Fault message with proper fault
                  string</expected_output>
                  <quality_assessed1>Platform
                  robustness</quality_assessed1>
                  <quality_assessed2>Platform
                  security</quality_assessed2>
                  <quality_assessed3>Platform fault
                  tolerance</quality_assessed3>
                  </test_case>

                  <!-- More test cases here -->

                  <test_case>
                  <test_case_id>Max_Numeric</test_case_id>
                  <test_datatype>double</test_datatype>
                  <test_data>1.79769313486231570E308</test_data>
                  <expected_output>Response message</expected_output>
                  <quality_assessed1>WS implementation
                  robustness</quality_assessed1>
                  </test_case>
                  </test_cases>
               </input_part>
           </ordered_input_parameters>
           <output_message>squareRootResponse</output_message>
           <output_parameters>
               <output_part>
            <output_part_name>squareRootReturn</output_part_name>
            <output_part_dataType>double</output_part_dataType>
            </output_part>
           </output_parameters>
        </operation>
    </operations>
</web_service>
```

List 7.13. Test Cases for the Square Root Web Service

```
<?xml version="1.0" encoding="UTF-8"?>

<web_service>
   <service_name>SquareRootService</service_name>
   <operations>
     <operation>
       <operation_name>squareRoot</operation_name>
          <test_cases>
            <test_case>
              <in0>oilcqhiflyzzasgkzcplg</in0>
               <fault>
                <fault_code>Server.userException</fault_code>
                <fault_string>org.xml.sax.SAXException: Bad types
            (class java.lang.String -&gt; double)</fault_string>
                <fault_detail>
                  <hostname>e-sci035</hostname>
                </fault_detail>
               </fault>
            </test_case>
                <in0>INF</in0>
                <output>Infinity</output>
                <invokation_time>47 ms</invokation_time>
            </test_case>
            <test_case>
                <in0>-INF</in0>
                <output>NaN</output>
                <invokation_time>15 ms</invokation_time>
            </test_case>
            <test_case>
                 <in0>NaN</in0>
                  <output>NaN</output>
                  <invokation_time>47 ms</invokation_time>
            </test_case>
            <test_case>
                    <in0>1.7976931348623157E308</in0>
                    <output>1.3407807929942596E154</output>
                    <invokation_time>31 ms</invokation_time>
            </test_case>
            <test_case>
                    <in0>1.7976931348623156E308</in0>
                    <output>1.3407807929942596E154</output>
                    <invokation_time>31 ms</invokation_time>
            </test_case>
                <!--more test cases and responses here -->
          </test_cases>
     </operation>
   </operations>
</web_service>
```

List 7.14. Test Cases with Responses for the Square Root Web Service

Table 7.10. Test Data and Responses for the Square Root Web Service

Test Case ID	Test Data	SOAP response of fault
String_Replacement	oilcqhiflyzzasgkzcplg	FMP
Date-Time_Replacement	2008-05-01	FMP
Boolean_Replace-ment	true	FMP
null_Replacement	null	FMI
Max_Numeric1	INF	INF
Max_Numeric2	1.7976931348623157 E308	1.3407807929942596E154
Less_Max	1.7976931348623156 E308	1.3407807929942596E154
Min_Value1	-INF	NaN
Min_Value2	4.9E-324	2.2227587494850775E-162
Above_Min	4.8E-324	2.2227587494850775E-162
Divide_by_Zero	0	0
Empty_String	Empty string	FMP
NaN_Replacement	NaN	NaN

7.6 Assessing Platform Robustness

The Web Services examples in Section 7.2 and 7.3 were all deployed in the Axis platform which is hosted in a Tomcat Server. To assess the effect of the Web Service platform of the Web Services robustness, one of the Web Services examples in section 7.2 was deployed in the GLUE platform. After that the response or fault SOAP message generated by GLUE was compared with those generated by Axis for the same WS-Robust test cases.

7.6.1 Configuration

The implementation of this Section example uses Java version 1.5.0_06, Axis 1.4, GLUE 1.2, Tomcat 6 Server and HTTP Server.

7.6.2 Scenario

In Section 7.2, forty one Web Services, each accepting different primitive or derived from primitive datatype, were implemented and deployed in Axis. Instead of repeating all the examples in Section 7.2 using the GLUE platform, the equivalent partitioning testing was used for the datatype partitions in Table 5.2. For the Numeric datatypes class, the *double* datatype was chosen to represent datatypes in this class. In a similar way for the String datatypes, the *string* datatype was chosen, for the Date-Time datatypes, the *date* datatype was chosen to represent datatypes in this class, and finally for the Boolean datatypes, the *boolean* datatype, the only element in this class was used.

The same Web Services that accepts *double, string, date, and boolean* that were deployed in the Axis platform in Section 7.2, were deployed in the GLUE platform to compare the results with those obtained for Axis.

Tables 7.11 to 7.14 show the comparative responses using Axis and GLUE.

Table 7.11. Responses of Axis and GLUE for a Web Service with *double* Datatype

Test Case	Axis 1.4 *response* or *fault*	GLUE 1.2 *response* or *fault*
String_Replacement	FMP	FM with fault string: 'For input string <test>' the *detail* element that contained a stack trace
Date-Time_Replacement	FMP	FM with fault string: 'For input string <2007-02-20T00:00:00.000Z > the *detail* element that contained a stack trace
Booleant_Replacement	FMP	FM with fault string: 'For input string <true > the *detail* element that contained a stack trace
Null_Replacement	FM with fault string: 'No such operation'	FM with empty fault string the *detail* element that contained a stack trace

Table 7.12. Responses of Axis and GLUE for a Web Service with *string* Datatype

Test Case	Axis 1.4 *response* or *fault*	GLUE 1.2 *response* or *fault*
Numeric_Replacement	FMP	RM (Numeric value accepted)
Date-Time_Replacement	FMP	RM (Date-Time value accepted)
Booleant_Replacement	FMP	RM (Boolean value accepted)
Null_Replacement	RM (null accepted)	RM (null value accepted)

Table 7.13. Responses of Axis and GLUE for a Web Service with *date* Datatype

Test Case	Axis 1.4 *response* or *fault*	GLUE 1.2 *response* or *fault*
Numeric_Replacement	FMP	RM with a null replacing the Numeric value passed*
String_Replacement	FMP	RM with a null replacing the String value passed
Booleant_Replacement	FMP	RM with a null replacing the Boolean value passed
Null_Replacement	RM (null accepted)	RM (null accepted)

* List 7.15 is the SOAP response for this test case

Table 7.14. Responses of Axis and GLUE for a Web Service with *boolean* Datatype

Test Case	Axis 1.4 *response* or *fault*	GLUE 1.2 *response* or *fault*
Numeric_Replacement	FMP	FMP
String_Replacement	FMP	FMP
Date-Time_Replacement	FMP	FMP
Null_Replacement	RM (null value accepted)	RM (null accepted)

```
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/">
 <soap:Body>
  <n:dtRetResponse xmlns:n="http://tempuri.org/convert.dt.DateReturn">
  <Result xsi:type="xsd:string">The Date passed is null</Result>
 </n:dtRetResponse>
 </soap:Body>
</soap:Envelope>
```

List 7.15. The SOAP *response* message produced by GLUE for Numeric_Replacement test case with a *date* datatype

7.6.3 Results

The results obtained for the four examples in Table 7.11 through Table 7.14 will be discussed separately as follows:

1) The Web Service that accepts primitive ***double*** input datatype (Table 7.11):

- Axis is robust when applying the *String_Replacement* test case while GLUE is not. The reason is that GLUE returned a *fault* message with a *fault string* that does not describe the fault that happened, the fault string is "*For input string <test>*" where *test* is the *string* that is used to replace the actual *double* datatype.
- Axis is robust when applying the *Date-Time_Replacement* test case while GLUE is not. The reason is that GLUE returned a *fault* message with a *fault string* that does not describe the fault that happened, the *fault string* is "*For input string <2007-02-20T00:00:00.000Z>*" where *2007-02-20* is the *date* that is used to replace the actual *double* datatype.
- Axis is robust when applying the *Boolean_Replacement* test case while GLUE is not. The reason is that GLUE returned a *fault* message with a *fault string* that does not describe the fault that happened, the *fault string* is "*For input string <true>*" where *true* is the *Boolean* that is used to replace the actual *double* datatype.
- Both Axis and GLUE are not robust when applying the *Null_Replacement* test cases generation rule that replace the input parameter with *null*. Axis is not robust because it returned a *fault* message with a *fault string* that does not describe the fault that happened, while GLUE is not robust because it returned an empty fault message which means that the Service Requester will not know what that fault that has happened.

- GLUE was not secure when applying the *String_Replacement, Date-Time_Replacement, Boolean_Replacement,* and *Null_Replacement*. The reason is the GLUE returned a SOAP fault message with a stack trace inside the detail element of this message.
- Axis was secure in all the test cases applied.

2) The Web Service that accepts primitive ***string*** input datatype (Table 7.12):

- Axis is robust when applying the *Numeric_Replacement, Date-Time_Replacement,* and *Boolean_Replacemnet* test case while GLUE is not. The reason is that GLUE returned a *SOAP response* message while a SOAP fault with proper fault string was expected in these test cases.
- Both Axis and GLUE are not robust when applying the *Null_Replacement* test cases generation rule that replace the input parameter with *null*. The reson is that both Axis and GLUE accepted the null input and did not return a SOAP fault.

3) The Web Service that accepts primitive ***date*** input datatype (Table 7.13):

- Axis is robust when applying the *Numeric_Replacement, String_Replacement,* and *Boolean_Replacemnet* test case while GLUE is not. The reason is that GLUE returned a *SOAP response* message where the Numeric, String, and Boolean inputs are converted to null. List 7.12 represents the SOAP response when a Numeric value replaced the *date* input. The GLUE platform passed a *null* value, to the Web Service implementation (See Fig. 2.4), instead of the Numeric value (*integer*) that was passed in the SOAP request, this is clear from the response of the Web Service operation *"The Date passed is null"* (See List 7.12).
- Both Axis and GLUE are not robust when applying the *Null_Replacement* because both Axis and GLUE accepted the null input and did not return a SOAP fault.

4) The Web Service that accepts primitive ***boolean*** input datatype (Table 7.14):

- Both Axis and GLUE are robust when applying the *Numeric_Replacement, String_Replacement,* and *Date-Time_Replacement* test case because both Platforms behave as expected by these test cases by sending a SOAP fault with proper fault string that describe the fault happened of changing the input datatype.
- Both Axis and GLUE are not robust when applying the *Null_Replacement* because both Axis and GLUE accepted the null input and did not return a SOAP fault.

Table 7.16 summarizes the results obtained by showing the number of the Test Cases, the number of Robustness Failures, Security Failures, and the description of the fault the led to these failures for the Axis and GLUE Web Services platforms.

Table 7.16. Comparison of Robustness and Security between Axis and GLUE

Platform	Test Cases	Robustness Failures	Security Failures	Fault Description
Axis	16	4	0	Handling null input
GLUE	16	13	4	Handling changing the input datatype Handling null input Stack trace in the SOAP fault

The conclusion is that Axis is more robust and secure than GLUE because, for the same test cases, GLUE caused 13 robustness failures and 4 security failures while Axis caused only 4 robustness failures and no security related failures. Figure 7.1 also summarizes these results.

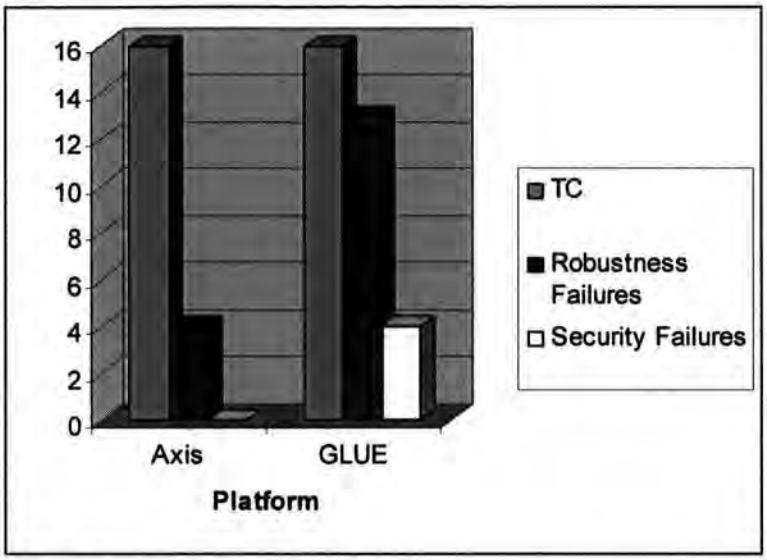


Fig. 7.1. A Comparison of Robustness and Security between Axis and GLUE

7.7 Summary

This Chapter has evaluated the effectiveness of the Web Services robustness testing framework that uses test case generation rules defined in Chapter 5. It has also shown that an efficient tool can be developed that applies the framework to the following examples or case studies:

- Forty one Web Services that accepts different primitive XML Schema datatypes as input.
- A Web Service that accepts a user-derived datatype
- The Amazon Web Service, a commercially available Web Service.
- A Web Service that was developed as part of a research project that is based on a remote system.

A comparison of the robustness of the Axis and GLUE platforms has been made using the test cases rules, and this caparison has revealed that Axis is more robust than GLUE for the example that have been used.

These examples have demonstrated that the Web Services robustness testing framework and WS-Robust is able to assess the robustness of a Web Service implementation and platform. Also it has been shown that Axis does not have a validation for the constraints of the input parameter, which means that it does not check if the input parameter satisfies the constraints described by the WSDL's type element.

Chapter 8

Conclusion and Future Work

8.1 Introduction

Web Services are still not widely used because Service Requesters do not trust Web Services that were built by others. To solve this problem all the trustworthiness requirements such as reliability, safety, security, interoperability must be addressed by researchers and practitioners.

After a survey on the field of Web Services testing and quality attributes, it has been found that most of the research has been done to test if the Web Service operation satisfy the Service Requester requirements. This type is testing is called validation testing. Very little research used fault-based testing with Web Services and these works did not specify the quality attribute being assessed. The research in this thesis is different than the previous work because:

- It provides a systematic way of generating test cases to assess the robustness of Web Services.
- It automates the process of test case generation based on WSDL
- It automates the process of test client generation

Test cases in this thesis are based on the XML Schema input parameter specification inside WSDL and the robustness faults that may affect a Web Service are based on violating these specifications. Assessing the robustness quality attribute contributes to

the assessment of other quality attributes such as security and fault tolerance to wrong input.

A proof of concept tool has been implemented that can help the Service Requester to assess the robustness of a Web service based only on its WSDL.

The robustness of a Web Service may be affected by the Web Service platform or the middleware that this Web Service is deployed on. The test cases designed in this thesis distinguish between testing the robustness of the platform and testing the robustness of the Web Service implementation. When the test data in a test case is valid then this test case is suppose to assess the robustness of the Web Service implementation because the platform should not intercept the SOAP that contains this test data. However, when the test data is invalid for example changing the datatype of the input parameter, then the platform robustness is being tested. This is because the platform must check the input parameter datatype and not send this invalid data to the Web Service implementation.

8.2 Contributions

This Section will discuss how this thesis has achieved its contributions that were introduced in Chapter 1.

1. ***Developing an approach to assess the robustness quality attribute of Web Service based only on the specification of the operations' input parameters datatypes inside the WSDL document of the Web Service under test:***

   This thesis has introduced an approach in Chapter 5 that can be used to assess the robustness and other related quality attributes based only on the input

This thesis has introduced an approach in Chapter 5 that can be used to assess the robustness and other related quality attributes based only on the input parameters datatype specification inside WSDL. Since WSDL use XML Schema datatypes to achieve interoperability, analysis has been done on each of the three categories of these datatypes, namely, primitive, user-derived, and complex. For each of these categories, Chapter 5 specified how the test cases can be generated and also what quality attribute, fault, testing techniques, and WSDL component are related to each test case.

2. ***Detecting robustness and security faults in Web Services implementations and platforms:***

   In Chapter 7, the approach developed in this thesis was able to detect robustness and security faults in experimental Web Services and also in a real commercial Web Service (Amazon).

3. ***Analysis of which faults affect the robustness quality attribute of Web Services:***

   The test case generation rules in Chapter 5 are considered a schema for describing the faults that affect the robustness quality attribute of a Web Service. The rules in Section 5.4 showed how a single fault may affect more than one quality attribute that are related to robustness.

4. ***Implementing a prototype tool that demonstrates the feasibility of the proposed Web Services robustness testing approach. The tool is able to automatically***

generate test cases to assess the robustness of Web Service and to automatically write a test client depending on the generated test cases.

The approach that was introduced in Chapter 5 had been implemented in the WS-Robust tool that is introduced in Chapter 6.

Chapter 7 has demonstrated that WS-Robust can automatically generate test cases depending on WSDL. WS-Robust automatically generated test cases, depending on the test case generation rules. Section 7.2 described the forty one Web Services that accept inputs of the different primitive XML Schema datatypes specified in Fig. 2.5. Test cases for user-derived and complex datatypes were generated and described in Section 7.3 and 7.4.

Chapter 6 showed the details of the implementation of the test client that can automatically invoke the Web Service under test using the Test Cases document that is generated by WS-Robust. However, the automation of the test client generation process was possible for Web Services that accepts an input of primitive, derived from primitive and user-derived datatypes but not for Web Service that accepts a complex datatype. The reason for this is that other programs, such as *wsdl2java*, are needed to generate the client in case of complex datatypes. For this reason, the automation of test case generation and client generation for the Web Services with complex input datatype will be carried out as part of the future work.

5. ***Analyze the effect of the Web Service platform on the robustness and security quality attributes.***

   Section 7.5 in Chapter 7 showed how the Web Services platform may affect the robustness and security by comparing Axis and GLUE Web Services platforms. For the experimental examples used, Axis has less robustness faults than GLUE. The GLUE platform showed some security faults while Axis did not.

8.3 Future Work

Future work is needed in the following directions:

- Assessing other quality attributes of Web Services: The test case generation schema in Chapter 5 showed the faults that are related to the robustness and other related quality attributes, if analysis on the faults that affect the other quality attributes in Fig. 3.1 such as safety and availability, then we can reach a better assessment of the trustworthiness of Web Services and increase their use.
- Since testing Web services is expensive we want to find a way to reduce the number of test cases but without compromising the robustness assessment: When finding the test cases for the Web Service operations with more than one input parameter, this thesis approach used the cross product of the test cases for each parameters. However, this method will produce a lot of test cases specially if an operation has many parameters. Also, it has been noticed that platforms stop when detecting the first fault, which means that the first occurring fault will hide the other faults in the input parameters. So, for this reason and to reduce the cost of test, the future work will modify WS-Robust to make each invocation to the Web

Service has only one invalid input and the other inputs are valid instead of using the cross product.

- Automate the process of client generation for Web Service with complex datatypes: Test client generation for Web Service with complex datatypes was done manually; future work will automate this process.
- Test Case generation when the input parameter of *list*, *union* (See Section 2.6.2.1) datatype: This thesis handled test case generation when the input parameter is of primitive, user-derived, or complex datatypes only, however, the input parameter might also be of list or union datatype.
- Test case generation when the input parameter is an array of other datatypes: This thesis did not handle the case when the input parameter is an array of simple, user-derived, or complex datatypes.
- Test case generation when the message exchange pattern is not Request-Response: There are four types of message exchange in Web Service (See Section 2.6.3), but this thesis only considered Request-Response.
- Finding test case generation rules when the user-derived datatype has the *pattern, enumeration, whitespace, totalDigits, and fractionDigit* constraining facets: Test case generation rules did not consider these constrains.
- Analyzing how other elements of WSDL may affect the robustness quality attribute: In this thesis approach, only the input parameter XML Schema datatypes are manipulated, future work will assess the affect of manipulating other WSDL elements such as *binding* (See Fig. 2.6) on Web Services robustness.
- Finding a method to inform the Service Provider how to modify their WSDL to increase Web Service testability: It has been noticed that real Web Services, such

as the Amazon Web Service use only primitive datatypes and also describe all their parameters as a *string* even though they should have a numeric or a date datatype, WS-Robust can be modified so that it can send a message to the Service Provider to suggest to them what changes they should make to the datatype specifications such as changing the datatype of a parameter or adding some constraint facets to it.

- Analyzing if there exist other faults that also may affect the robustness quality attribute of Web services: This thesis addressed the faults that are addressed in the test cases generation rules in Chapter 5, future work will try to investigate more in the testing literature for other faults that may affect Web Services robustness and then add more test cases to detect such faults.

8.4 Summary

The main contribution of this thesis is providing a framework and a tool to assess the robustness quality attribute of Web Services and increase the Service Requester and Provider trustworthiness of Web Services. However, the approach in this thesis did not give a complete assessment of the trustworthiness, if more research is done in this field and if Service Providers and Requesters add more test cases generation rules depending on there experience in different domains of Web Services, then these new test cases will address the other quality attributes and the trustworthiness and usage of Web Services will increase. Web Services will then become the dominant distribution systems architecture.

Bibliography

Adrion W., Branstad M. & Cherniavsky, J. (1982) Validation, Verification, and Testing of Computer Software. ACM Computing Surveys, (Vol. 14, No. 2), pp.159-192.

Amazon. (2007). Amazon Web Service.

[Retrieved from: http://www.amazon.com/AWS-home-page-Money/b/102-8408021-2173714?ie=UTF8&node=3435361]

Accessed on July 2007.

Apache Software Foundation. (2005) Web Services – Axis, Client-side Axis.

[Retrieved from http://ws.apache.org/axis/java/client-side-axis.html]

Accessed September 2007.

Apache Software Foundation. (2006) Apache Tomcat 6.0.

[Retrieved from http://tomcat.apache.org/tomcat-6.0-doc/index.html]

Accessed September 2007.

Apache Software Foundation. (2007). Web Services – Axis, Apache web Services Project

[Retrieved from http://ws.apache.org/axis/]

Accessed July 2007.

Avizienis, A., Laprie, J.-C., Randell, B. & Landwehr, C. (2004) Basic Concepts and Taxonomy of Dependable and Secure Computing. IEEE Transactions on Dependable and Secure Computing, (Vol. 1, No. 1), China, pp. 11-33.

Bibliography

Bai, X. & Dong, W. (2005) WSDL-Based Automatic Test Case Generation for Web Services Testing. Proceedings of the 2005 IEEE Workshop on Service-Oriented System Engineering (SOSE'05), pp. 207-212.

Bass, L., Clements, P. & Kazman, R. (2003). Software Architecture in Practice, Second Edition. ISBN: 0-321-15495-9. Addison Wesley.

Beizer, B. (1990) Software Testing Techniques, Second Edition. Van Nostrand Reinhold, New York, USA, ISBN 0-442-20672-0.

Bloomberg, J. (2002). Testing Web Services Today and Tomorrow. Rational Edge e-zine for the Rational Community, (2002)

[Retrieved from

http://www.ibm.com/developerworks/rational/library/content/RationalEdge/oct02/WebTesting_TheRationalEdge_Oct02.pdf]

Accessed October 2007.

Boehm, B. W., Brown, J. R. & Lipow, M. (1976) Quantitative Evaluation of Software Quality. Proceedings of the 2nd International Conference on Software Engineering (ICSE), California, USA, 1976, pp. 592-605.

Canfora, C. (2005). User-side Testing of Web Services. Proceedings of the IEEE Ninth European Conference on Software Maintenance and Reengineering (CSMR'05), 21-23 March, Manchester, UK, pp. 301-301.

Cerami, E. (2002) Web Services Essentials, Distributed Application with XML-RPC, SOAP, UDDI & WSDL. ISBN: 978-0596002244. O'Reilly.

Cohen, J., Plakosh, D. & Keeter, K. (2005). Robustness Testing of Software-Intensive Systems: Explanation and Guide. Technical Note CMU/SEI-2005-TN-015, Software Engineering Institute, Carnegie Mellon University, USA.

Bibliography

Csallner, C. & Smaragdakis, Y. (2000) JCrasher: an automatic robustness tester for Java. Software-Practice & Experience, (Vol. 34, No. 11), September, 2004. pp. 1025-1050.

Curbera, F., Duftler, M., Khalaf, R., Nagy, W., Mukhi, N. and Weerawarana, S. (2002) Unravelling the Web Services Web, An Introduction to SOAP, WSDL, and UDDI. IEEE Internet Computing, (Vol. 6, No. 2), pp. 86-93.

Deitel, H. M., Deitel, P.J., Gadzik, J.P., Lomeli, K., Santry, S.E. & Zhang. S. (2003). Java Web Services for Experienced Programmers. ISBN: 0-13-046134-2, Prentice Hall.

Dix, M. & Hofmann, H.D. (2002) Automated Software Robustness Testing - Static and Adaptive Test Case Design Methods. Proceedings of the IEEE 28th Euromicro Conference, 4th-6th, September, Germany, pp.62-66.

Empirix (2007) e- Test Suit for Web Services: Web Services Testing and Monitoring. [Retrieved from: http://www.empirix.com/products-services/w-testing.asp] Accessed November 2005.

Englander, R. (2002) Java and SOAP. ISBN: 0-596-00175-4, O'REILLY.

Erl, T. (2006) Service-Oriented Architecture, Concepts, Technology, and Design. ISBN: 0-131-85858-0, Prentice Hall.

Ferris, C. & Farrell, J. (2003) What Are Web Services? Communications of the ACM (Vol.46, No.6) June 2003, p. 31.

Fu, C., Ryder, B.G., Milanova, A. & Wonnacott, D. (2004) Testing of Java Web Services for Robustness. In Proceedings of the ACM International Symposium on Software Testing and Analysis, (ISSTA 2004) Massachusetts, USA, pp.23-34.

Bibliography

Garvin, D. (1984) What does 'Product Quality' Really Mean? Sloan Management Review, (Vol. 26, No. 1), pp. 25-45.

Gosh, A., Shah, V. & Schmid, M. (1998). An Approach for Analyzing the Robustness of Windows NT Software. Proceedings of the 21st National Information Systems Security Conference, Crystal City, USA, pp. 383-391.

Gourley, D., Totty, B., Sayer, M., Aggarwal, A. & Reddy, S. (2002) HTTP: The Definitive Guide, ISBN: 1-565-92509-2, O'Reilly.

Graham, S., Davis, D., Simenov, S., Daniels, G., Brittenham, P., Nakamura, Y., Fremantle, P., Konig, D. & Zentner, C. (2005). Building Web Services with Java, Making sense of XML, SOAP, WSDL, and UDDI. Second Edition. ISBN: 0-672-32641-8, Sams Publishing.

Harold, E.R. & Means, W.S. (2004) XML IN A NUTSHELL, A desktop Quick Reference, , third edition, ISBN: 0-596-00764-7, O'REILLY, USA.

Harrold, M. (2000) Testing: A Roadmap. Proceedings of the IEEE 22nd International Conference on the Future of Software Engineering, June 2000, Ireland, pp.61-72.

Hetzel, W. (1973) Programme Test Methods. ISBN: 978-0137296248. Prentice Hall.

Hsueh, M., Tsai, K. & Iyer, R.K. (1997) Fault Injection Techniques and Tools. IEEE Computer (Vol. 30, No. 4) April 1997. pp. 75-82.

Huhns, M. & Singh, M. (2005) Service-Oriented Computing: Key Concepts and Principles. IEEE Internet Computing, (Vol. 9, No. 1) January-February 2005.

IBM. (2006) New to SOA and Web Services.

[Retrieved from

http://www.ibm.com/developerworks/webservices/newto/websvc.html]

Accessed September 2006.

Bibliography

IEEE (1990) IEEE Standard Glossary of Software Engineering Terminology. IEEE Computer Society, Std 610.12-1990.

IEEE (1995) IEEE Guide to Classification for Software Anomalies. Software Engineering Standards Committee of the IEEE Computer Society, IEEE Computer Society, Std 1044.1-1995.

ISO 9126-1: 2001 (2001) Software Engineering – Product quality – Part 1: Quality Model, International Organization of Standardization, Geneva, Switzerland.

Jorgensen, P. C. (2002) Software Testing, A Craftsman's Approach Second Edition. ISBN 0-8493-0809-7, CRC Press.

Koopman, P., Sung, J., Dingman, C., Siewiorek, D. & Marz, T. (1997). Comparing Operating Systems Using Robustness Benchmarks. In Proceedings of the 16th IEEE Symposium on Reliable Distributed Systems, October, 22-24, North Carolina, USA, pp. 72-79.

Korel, B. (1999) Black-Box Understanding of COTS Components. Proceedings of the IEEE 7th International Workshop on Program Comprehension (IWPC), 5-7 May 1999, Pennsylvania, USA, pp.92-99.

Leavitt, N. (2004) Are Web Services Finally Ready To Deliver? IEEE Computer, (Vol. 37, No. 11), pp.14-18.

Looker, N., Munro, M. & Xu, J. (2004). Simulating Errors in Web Services. International Journal of Simulation: Systems, Science & Technology, (Vol. 5, No. 5), pp. 29-37.

Looker, N., Munro, M. & Xu, J. (2007). Determining the Dependability of Service-Oriented Architectures. International Journal of Simulation and Process Modelling (Vol. 3, No. 1/2), pp. 88-97.

Bibliography

Lyndsay, J. (2003). A Positive View of Negative Testing. Workroom Production Ltd. London, UK.

[Retrieved from http://www.workroom-productions.com/papers/PVoNT_paper.pdf] Accessed October, 2007.

Marsden, E., Fabre, J.-C. & Arlat, J. (2002). Dependability of CORBA Systems: Service Characterization by Fault Injection. Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems (SRDS'02), October, 13-16, Japan, pp.276-285.

McCall, J. A., Richards, P. K. & Walters, G. F. (1977) Factors in Software Quality. Three volumes, US Department of Commerce, National Technical Information Service (NTIS).

Mercury (2007) Service Test.

[Retrieved from: http://www.mercury.com/us/products/quality-center/functional-testing/service-test/],

Accessed October 2007.

Miller B.P., Fredriksen, L. & So, B. (1990). An Empirical Study of the Reliability of UNIX Utilities. Communications of the ACM, (Vol. 33, No. 12) pp. 32-44.

Murnane, T., Hall, R., & Reed, K. (2005) Towards Describing Black-Box Testing Methods as Atomic Rules. Proceedings of the 29th Annual International Computer Software and Applications Conference (COMPSAC'05), 26-28 July, Edinburgh, Scotland, pp. 437-442.

Murnane, T., Reed, K. & Hall, R. (2006) Tailoring of Black-Box Testing Methods. Proceeding of the IEEE 2006 Australian Software Engineering Conference (ASWEC'06), 18-21 April, Australia, pp. 292-299.

Bibliography

Myers, G. (1979). The Art of Software Testing. ISBN 0-471-04328-1, John Wiley.

Neumann, P. (2004). Principled assuredly trustworthy composable architecture. Emerging draft of the final report for DARPA's composable high assurance trustworthy systems (CHATS) program.

[Retrieved from http://www.csl.sri.com/users/neumann/chats4.pdf]

Accessed October 2007.

OASIS. (2004). UDDI Spec TC, UDDI Version 3.0.2. UDDI Spec Technical Committee Draft October 2004,

[Retrieved from: http://uddi.org/pubs/uddi-v3.0.2-20041019.htm].

Accessed October 2005.

Object Management Group. (1998). Common Object Request Broker: Architecture and Specification. Technical Report Revision 2.3, Object Management Group, Framingham, USA, July 1998.

Offutt, J. Xiong, Y. & Liu, S. (1999) Criteria for Generating Specification-based Tests. Proceedings of the 5th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '99), Las Vegas, USA, pp. 119-131.

Offutt, J. Liu, S. Abdurazik, A. & Ammann, P. (2003) Generating test data from state-based specifications. The Journal of Software Testing, Verification and Reliability, pp. 25-53.

Offutt, J. Xu, W. (2004). Generating Test Cases for web services Using Data perturbation. ACM SIGSOFT Software Engineering Notes, (Vol. 29, No. 5) September 2004, USA, pp. 1-10.

Osterweil, L. (1996). Strategic Directions in Software Quality. ACM Computing Surveys, (Vol. 28, No. 4), December 1996, pp. 738-750.

Bibliography

Pan, J., Koopman, P., Siewiorek, D., Huang, Y., Gruber, R. & Jiang, M. L. (2001) Robustness Testing and Hardening of CORBA ORB Implementation. Proceedings of The International Conference on Dependable Systems and Networks, 1-4 July, Goteborg, Sweden, pp.1-10.

Parasoft .(2007). Web Services Testing, SOA Development: Parasoft SOATest. [Retrieved from: http://www.parasoft.com/jsp/products/home.jsp?product=SOAP&itemId=101].

Accessed July 2007.

Parnas, D., Schouwen, v.J. & Kwan, S.P. (1990) Evaluation of Safety-Critical Software. Communication of the ACM (Vol. 33, No. 6) June 1990, pp. 636- 648.

Red Gate. (2007). Advanced .NET Testing System (ANTS).

[Retrieved from: http://www.red-gate.com/products/ants_load/index.htm],

Accesses October 2007.

Raghavan, G. (2002). Improving Software Quality in Product Families therough Systematic Reengineering. Proceedings of the Software Quality- ESSQ conference, Berlin, Germany, Springer-Verlag, pp. 90-99.

Schmid, M. & Hill, F. (1999). Data Generation Techniques for Automated Software Robustness Testing. Proceedings of the 16th International Conference on Testing Computer Software (ICTCS'99), Washington, USA.

Shelton, C., Koopman, P. & DeVale, K. (2000). Robustness Testing of the Microsoft Win32 API. Proceedings of the IEEE 2000 International Conference on Dependable Systems and Networks, June, 25-28, New York, USA, pp. 261-271.

Siblini, R. & Mansour, N. (2005). Testing Web Services. Proceedings of the 3rd ACS/IEEE International Conference on Computer Systems and Applications (AICCSA'05), 3-6, January, Cairo, Egypt, pp. 135-143.

Singh, M. & Huhns, M. (2005). Service-Oriented Computing: Semantics, Processes, Agents. ISBN: 978-0-470-09148-7, John Wiley & Sons Ltd.

Sommerrville, I. (2004). Software Engineering, Seventh Edition. ISBN: 0-321-21026-3. Addison-Wesley.

Toth, A., Varro, D. & Pataricca, A. (2003) Model-Level Automatic Test generation for UML State charts. 6th IEEE workshop on Design and Diagnostics Circuits and System DDECS, 14-16 April, Poznan, Poland, pp. 293-294.

Tsai, W.-T., Paul, R., Wang, Y., Fan, C. & Wang, D. (2002). Extending WSDL to Facilitate Web Services Testing. Proceedings of the 7th IEEE International Symposium on High Assurance Systems Engineering (HASE'02), Tokyo, Japan, pp. 171-172.

Tsai, W.T., Paul, R., Cao, Z., Yu, L., Saimi, A. & Xiao, B. (2003). Verification of Web Services Using an Enhanced UDDI Server. Proceedings of the 8th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems, 15-17 January, Guadalajara, Mexico, pp. 131 - 138.

Tsai, W.T., Chen, Y. & Paul, R. (2005a) Specification-Based Verification and Validation of Web Services and Service-Oriented Operating Systems. Proceedings of the IEEE 10th International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS'05), 2-4 February, Arizona, USA, pp. 139 - 147.

Tsai, W.T., Wei, X., Chen, Y., Xiao, B., Paul, R., & Huang, H. (2005b) Developing and Assuring Trustworthy Web Services. Proceedings of the 7th International Symposium on Autonomous Decentralized Systems (ISADS) 4-6 April, China, pp. 43-50.

Vlist, V. (2002) XML Schema – The W3C's Object-Oriented Descriptions for XML. ISBN: 0-596-00252-1, O'REILLY.

Voas, J., McGraw, G. & Ghosh, A. (1996). Gluing Together Software Components: How Good is Your Glue? Proceedings of the Pacific Northwest Software Quality Conference, Portland, USA, pp.338-349.

Voas, J. (1997). Error Propagation Analysis for COTS Systems. Computing & Control Engineering Journal, (Vol. 8, No. 6), December, pp. 269 – 272.

Voas, J. M. & McGraw G. (1998a) Software Fault Injection, Inoculating Programs against Errors. ISBN: 0-471-18381-4, John Wiley.

Voas, J. (1998b) Certifying Off-the-Shelf Software Components. IEEE Computer (Vol. 31, No.6) June 1998. pp. 53 - 59.

W3C. (2001). Web Services Description Language (WSDL) 1.1. W3C Note 15 March, [Retrieved from: http://www.w3.org/TR/2001/NOTE-wsdl-20010315].
Accessed May 2005.

W3C. (2004a). Web Services Architecture. W3C Working Group Note 11 February 2004,
[Retrieved from: http://www.w3.org/TR/ws-arch/]
Accessed May 2005.

W3C. (2004b). XML Schema Part 0: Primer Second Edition. W3C Recommendation 28 October 2004,

[Retrieved from: http://www.w3.org/TR/2004/REC-xmlschema-0-20041028/]

Accessed April 2005.

W3C. (2004c). XML Schema Part 2: Datatypes Second Edition. W3C Recommendation 28 October 2004,

[Retrieved from: http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/]

Accessed on April 2005.

W3C. (2005). Document Object Model (DOM)

[Retrieved from http://www.w3.org/DOM/].

Accessed January 2006.

W3C. (2006). Extensible Markup Language (XML) 1.0 (Fourth Edition). W3C recommendation 16 August 2006,

[Retrieved from: http://www.w3.org/TR/xml/].

Accessed January 2007.

W3C. (2007) SOAP version 1.2 Part 0: Primer (Second Edition). W3C Recommendation 27 April 2007,

[Retrieved from: http://www.w3.org/TR/2007/REC-soap12-part0-20070427/]

Accessed August 2007.

WebMethods (2007) GLUE tool.

[Retrieved from http://www.webmethods.com/Developers/]

Accessed January 2007.

Xu, W., Offutt, J. & Luo, J. (2005). Testing Web Services by XML Perturbation. Proceedings of the 16th IEEE International Symposium on Software Reliability Engineering (ISSRE'05), 8-11 November, Illinois, USA, pp. 257-266.

Yu, W.D., Aravind, D. & Supthaweesuk, P. (2006). Software Vulnarability Analysis for Web Services Software Systems. Proceedings of the 11th IEEE Symposium on Computers and Communications (ISCC), 26-29 June, Sardinia, Italy, pp. 740-748.

Zhang, J. (2004a). An Approach to Facilitate Reliability Testing of Web Services Components. Proceedings of the IEEE 15th International Symposium on Software Reliability Engineering (ISSRE'04), 2-5 November 2004, Bretagne, France, pp.210-218.

Zhang, J., Zhang, L.-J. & Chung, J.-Y. (2004b). An Approach to Help Select Trustworthy Web Services. Proceedings of the IEEE International Conference on E-Commerce Technology for Dynamic E-Business (CEC-East'04), 13-15 September, Beijing, China, pp.84-91.

Zhang, J. (2005a). Trustworthy Web Services: Action for Now. IEEE, IT Pro, January-February 2005, (Vol. 7, No. 1), pp. 32-36.

Zhang, J. & Zhang, L.-J. (2005b) Criteria Analysis and Validation of the Reliability of Web Services-oriented Systems. Proceedings of the IEEE International Conference on Web Services (ICWS'05) 11-15 July 2005, Florida, USA, pp. 621-628.

Zhang, J. & Zhang, L.-J. (2005c) Web Services Quality Testing. International Journal for Web Services Research, (Vol. 2, No. 2), pp. 1-4.

Zimmermann, J. (2003). Web Services Reduce Costs but Slow to Grow. ZDNet, [Retrievedfrom: http://news.zdnet.co.uk/hardware/0,1000000091,2130802,00.htm?r=2] Accessed July 2007.